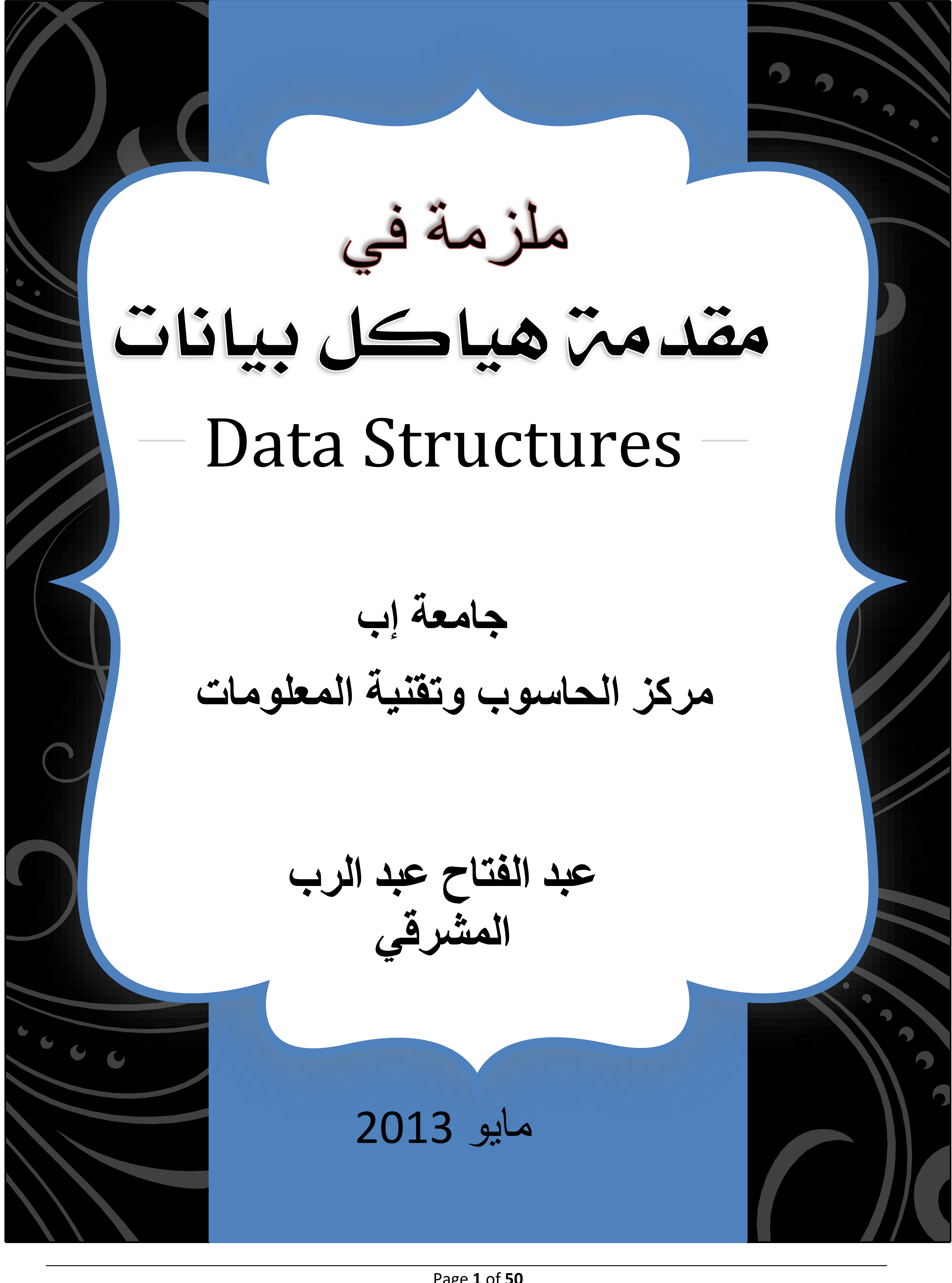

ملزمة في
مقدمة هياكل بيانات
Data Structures
جامعة إب
مركز الحاسوب وتقنية المعلومات
عبد الفتاح عبد الرب
المشرقي
مايو 2013


جامعة إب
مركز الحاسوب وتقنية المعلومات
إلى كل الأحبة خريجي وخريجات
الدفعة الثانية- برمجة حاسوب
دفعة
صناع الغد
اتقدم لكم باسمى معاني الحب
والامتنان على تقديركم العالي لي
واتمنى لكم مستقبلاً زاهراً
أ/ عبد الفتاح المشرقي

مقدمة في هياكل البيانات في+ + C

المرجع الأساسي في جمع هذه المادة:

INTODUCTION TO DATA STRUCTURES IN C++
Sanchit Karve
http://www.dreamincode.net/forums/topic/10157-data-structures-in-c-tutorial/

المحتويات

أولاً: الافتراضات ASSUMPTIONS

في هذا الشرح التدريبي سأفترض أن الجميع لديه معرفة عملية بكيفية استخدام الكلاسات classes في لغة C++، لأن جميع هياكل البيانات التي سوف نتعلمها الآن سوف تتم على أساس فهمنا للكلاسات. وأنا أدرك أن هناك الكثير من دروس وتمارين هياكل البيانات المتاحة، ولكن كل ما في الأمر أنه من النادر أن تجد أحداً يستخدم مفاهيم البرمجة الهدفية المعتمدة على الكائنات OOP في برمجة هياكل البيانات. لذلك سوف نركز هنا على توليد هياكل بيانات باستخدام الكلاسات. ويجدر التنويه إلى أن التعليمات البرمجية قد تم ترجمتها باستخدام بورلاند + + C ما لم يذكر خلاف ذلك.

ثانياً: مقدمة INTRODUCTION

فوائد هياكل البيانات Advantages of Data Structure

1. التحكم في توزيع البيانات والتعرف إلى طبيعتها وبنائها الأساسي بنسق معين في الذاكرة.
2. تمكين المبرمج من إبداع طرق مبتكرة لكتابة البرامج المختلفة.
3. اختصار زمن التخزين وزمن استرجاع البيانات من الذاكرة.
4. بناء برامج قوية ومتماسكة من حيث البناء والمنطق.

أنواع هياكل البيانات Data Structures Types:

1. هياكل بيانات استاتيكية (ثابتة) Static DS:
   a. المصفوفات Arrays والمتجهات Vectors.
   b. السلاسل الحرفية Strings.
   c. الجداول Tables.
   d. السجلات Records.

2. هياكل بيانات ديناميكية (متحركة) Dynamic DS:
   a. خطية Linear
      i. المكدس Stack
      ii. الطابور Queue
      iii. المجموعة Set
      iv. الملف File
      v. القائمة List
   b. غير خطية Non-Linear
      i. الشجرة Tree
      ii. المخطط Graph

وقد درسنا في أوقات ماضية بعض هياكل البيانات، وسنعمل في هذا المقرر على تغطية هياكل البيانات الأساسية التالية:

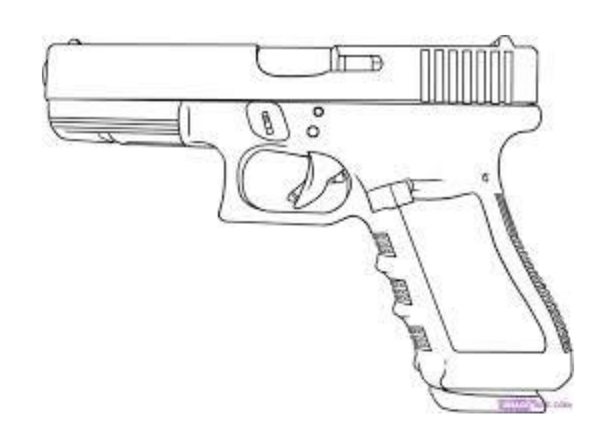
1. المكدس STACK
2. الطابور QUEUE
3. القائمة المتصلة LINKED LISTS
4. الأشجار الثنائية BINARY TREES

يجب علينا أيضاً الجمع بين هياكل البيانات معاً في وقت لاحق في هذا البرنامج التعليمي، مثل الجمع بين قائمة متصلة جنباً إلى جنب مع مكدس. يجب علينا أن نتعرف أيضاً عن القوائم المتصلة المضاعفة DOUBLY LINKED LISTS والقوائم المتصلة الدائرية CIRCULAR LINKED LISTS في هذا البرنامج التعليمي.
لذلك دعونا نبدأ دون إضاعة أي وقت.

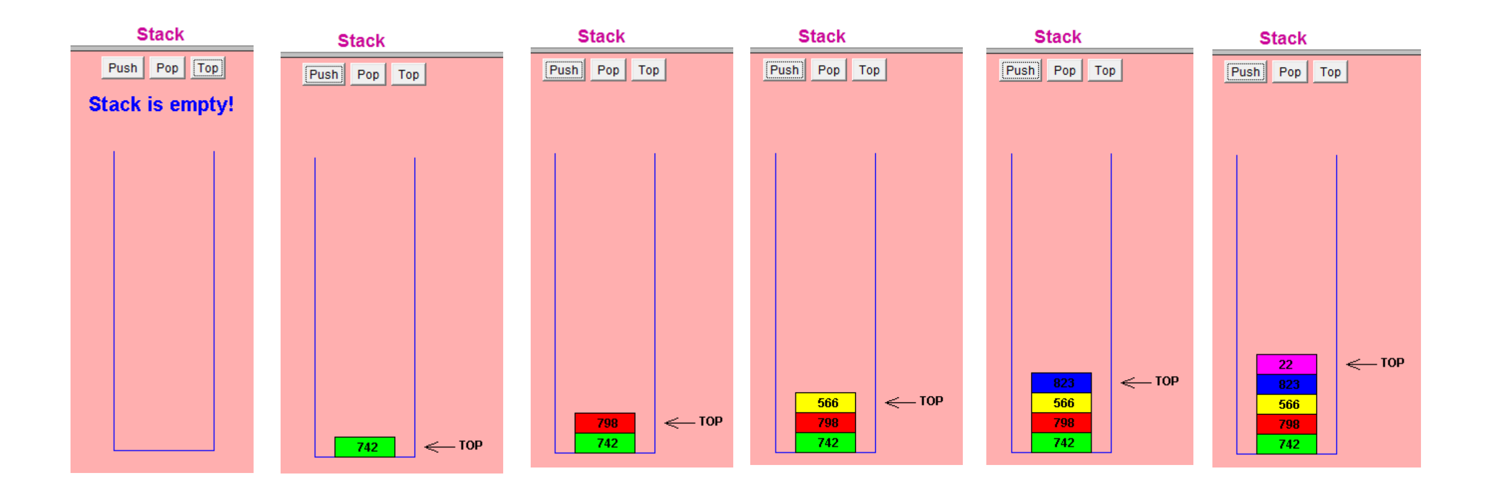
ثالثاً: المكدسات STACKS

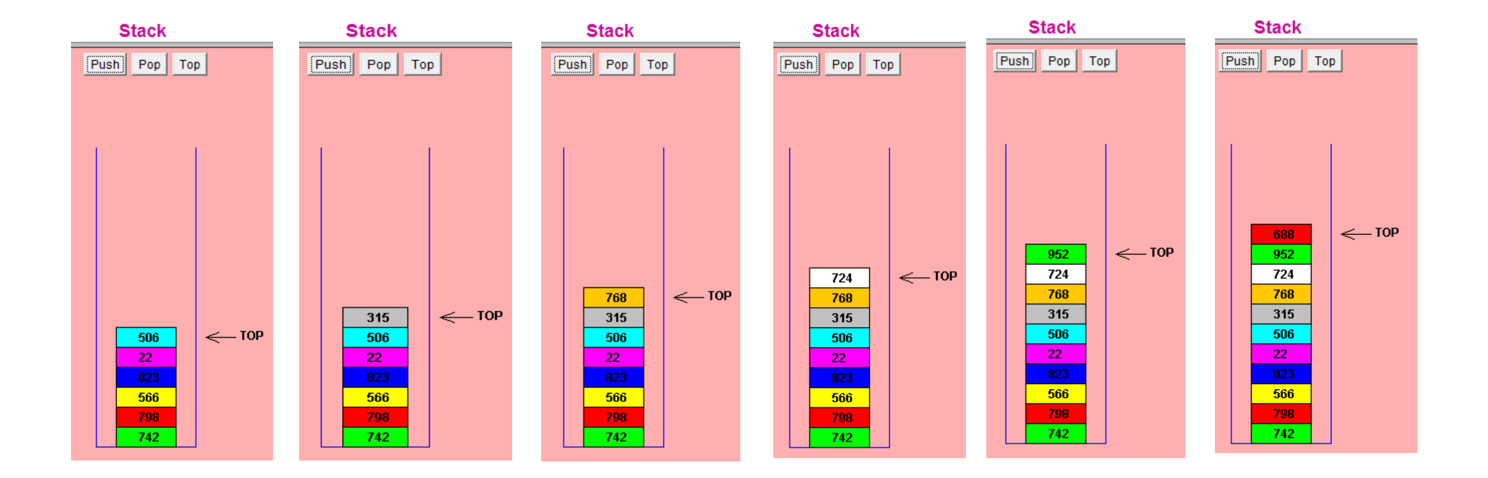
المكدسات هي هياكل بيانات يشيع استخدامها أثناء كتابة التعليمات البرمجية. وهذا المفهوم حقاً بسيط، مما يجعل كتابته أكثر سهولة. خذ بعين الاعتبار الموقف التالي: هناك كومة من 5 كتب على طاولة. وأنت تريد إضافة كتاب واحد إلى هذه الكومة. ماذا تفعل؟ يمكنك ببساطة إضافة الكتاب على الجزء العلوي من الكومة. ماذا إذا كنت ترغب في الحصول على الكتاب الثالث من كومة الكتب الجديدة التي تحوي ستة كتب؟ ستقوم حتماً برفع كل كتاب واحد تلو الآخر من جانب واحد وهو الجانب الأعلى من الكومة حتى يصبح الكتاب الثالث في الأعلى. حينها ستأخذ الكتاب الثالث وتعيد كل الكتب الأخرى إلى الكومة من خلال إضافتها إلى الكومة من الأعلى TOP.

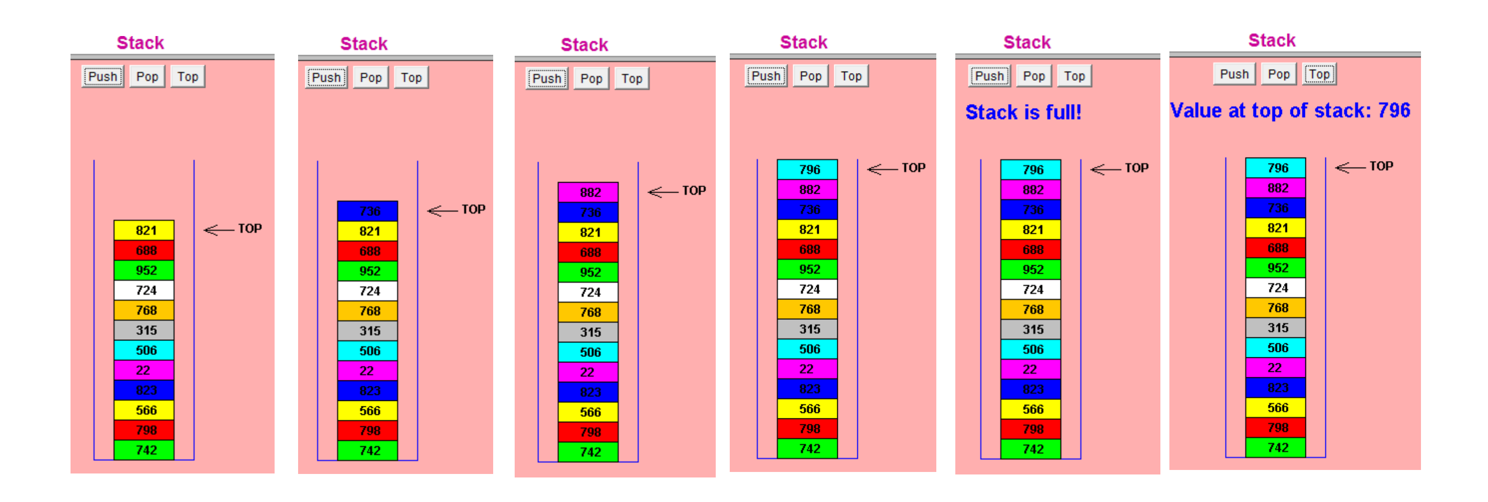
ومن الأمثلة الجيدة التي توضح فكرة عمل المكدس، خزنة السلاح. حيث أنها تمثل مكدساً لخزن حبات الرصاص بطريقة الداخل أولاً سيخرج أخيراً، وذلك لأن الخزنة تحتوي فقط على فتحة واحدة للإدخال والإخراج معاً.

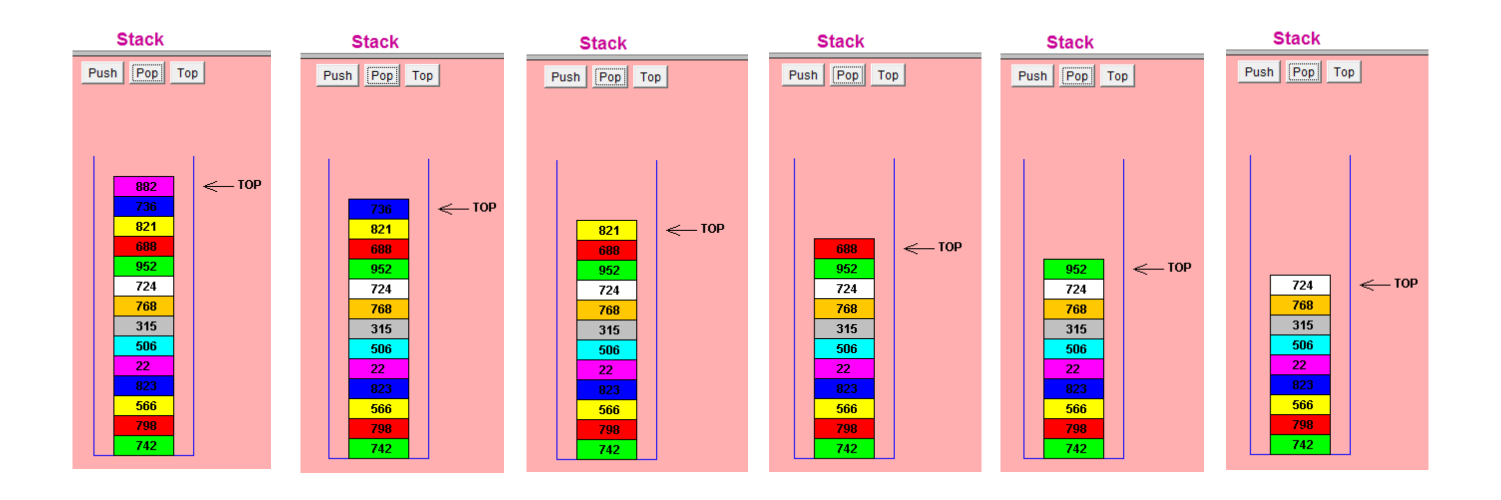


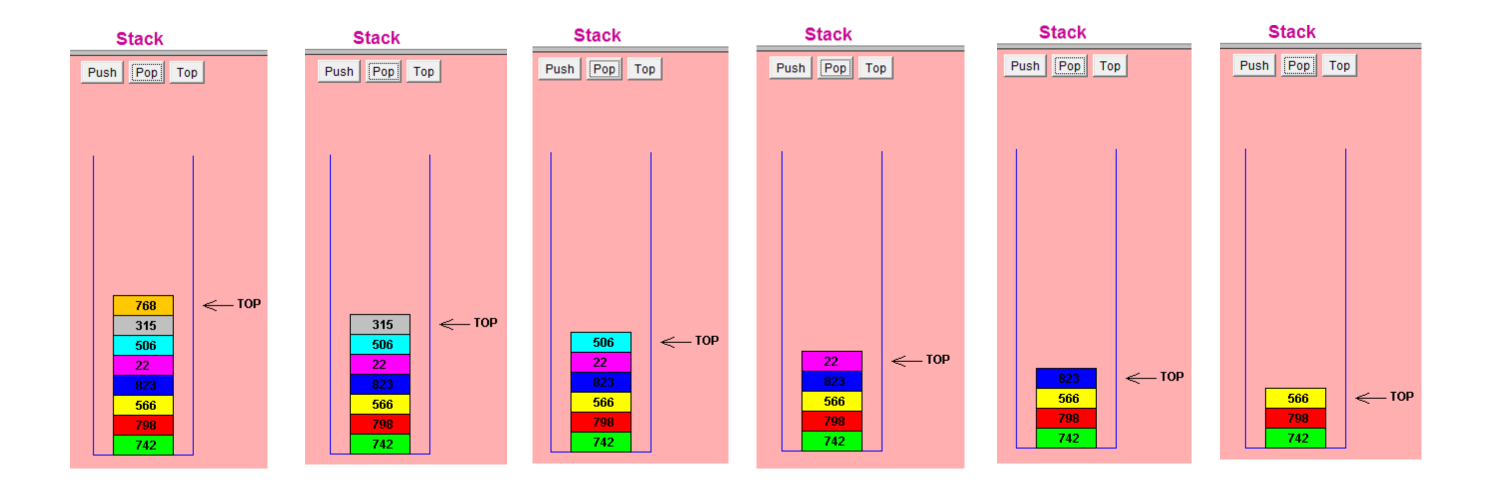
إذا كنت قد لاحظت، لقد أشرت إلى كلمة "الأعلى" (TOP) في سياق الحديث عدة مرات. نعم، أن TOP هي الكلمة الأكثر أهمية عند تعاملنا مع المكدس. يتم تخزين البيانات في المكدس حيث أن الإضافة للبيانات لا يسمح بها إلا من أعلى. إزالة أو حذف البيانات أيضاً تتم من أعلى المكدس. الآن قد تسأل أين تستخدم المكدسات؟ في الحقيقة، تستخدم المكدسات في كل المعالجات processors. فكل معالج لديه مكدس حيث يتم إدخال البيانات والعناوين إليه. ومرة أخرى يجب أن نتبع قاعدة TOP هنا أيضاً. المسجل ESP يضاف كمؤشر مكدس يشير إلى أعلى مكدس المعالج. على أي حال، فأن شرح كيفية عمل المكدس في المعالج أمر خارج عن حدود هذا الدرس التعليمي. دعونا الآن نقوم بكتابة هيكل بيانات المكدس. تذكر بعض المصطلحات المتعلقة بالمكدس قبل المتابعة. إن إضافة البيانات إلى المكدس يعرف أيضاً بإخراج البيانات popping، وحذف البيانات من المكدس يعرف أيضاً بإدخال البيانات pushing.

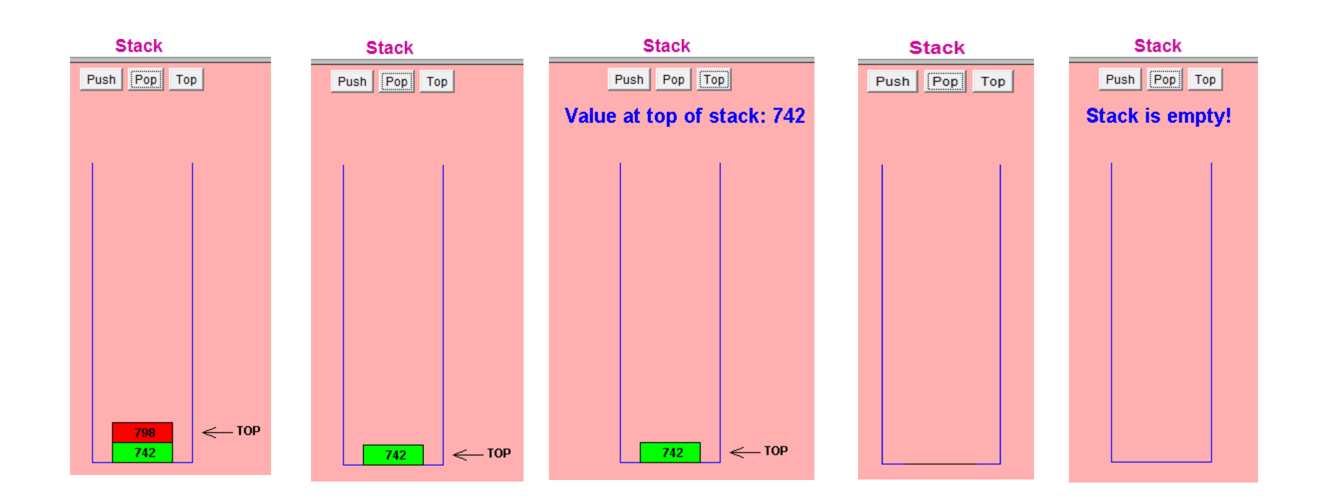



Stack
Push Pop Top
506 22 823 566 798 742
TOP
Stack
Push Pop Top
315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
688 952 724 768 315 506 22 823 566 798 742
TOP


Stack
Push Pop Top
821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
736 821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
882 736 821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
796 882 736 821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
Stack is full!
796 882 736 821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
Value at top of stack: 796
796 882 736 821 688 952 724 768 315 506 22 823 566 798 742
TOP


Stack
Push Pop Top
882 736 821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
736 821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
821 688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
688 952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
952 724 768 315 506 22 823 566 798 742
TOP
Stack
Push Pop Top
724 768 315 506 22 823 566 798 742
TOP





خوارزمية إضافة push عنصر a إلى المكدس:

1. زد مؤشر top بمقدار 1.
2. إذا كان top أقل من MAX قم بما يلي:
   a. ضع في الموقع top من المكدس القيمة a.
3. وإلا:
   a. اطبع عبارة "STACK IS FULL"
   b. أنقص top بمقدار 1.

خوارزمية الحذف pop من أعلى المكدس:

1. اختبر top، إذا كانت تساوي 1- قم بما يلي:
   a. اطبع عبارة "STACK IS EMPTY"
   b. اخرج من الخوارزمية
2. وإلا:
   a. ضع القيمة الموجودة في أعلى المكدس في متغير data.
   b. ضع في أعلى المكدس قيمة فارغة NULL.
   c. انقص top بمقدار 1.
   d. ارجع قيمة المتغير data.

```
#include <iostream>

using namespace std;

#define MAX 10          // أقصى محتوى للمكدس



class Stack
{
  private:
    int arr[MAX];    // مصفوفة تحوي كل البيانات
    int top;         //متغير يحوي موقع أعلى قيمة بيانات تم إدخالها إلى المكدس

  public:
      Stack()          //باني
      {
         top = -1;        //جعل قيمة الموقع الأعلى -1 في إشارة إلى خلو المكدس من البيانات
      }

     void push(int a);
     int pop();
};

void Stack::push(int a)   // دالة إضافة القيم إلى المكدس
      {
          top++;         // الزيادة بمقدار 1
          if(top < MAX)
           {
               //إذا كان هناك موقع شاغر في المكدس قم بتخزين القيمة في المصفوفة
               arr[top] = a;
           }
           else
           {
               cout << "STACK FULL!!" << endl;
               top--;
           }
      }

int Stack::pop()   //دالة حذف القيم من المكدس، حيث ترجع القيمة المحذوفة
     {
          if(top == -1)
          {
               cout << "STACK IS EMPTY!!!" << endl;
               return NULL;
          }
```

```
            else
            {
                 int data = arr[top]; //وضع قيمة توب في المتغير داتا
                 arr[top] = NULL;       //جعل الموقع الأصل فارغاً
                 top--;               // إنقاص توب بمقدار 1
                 return data;       // إرجاع العنصر المحذوف
            }
       }

int main()
{
 Stack a;
 a.push(3);
 cout << "3 is Pushed\n";
 a.push(10);
 cout << "10 is Pushed\n";
 a.push(1);
 cout << "1 is Pushed\n\n";

 cout << a.pop() << " is Popped\n";
 cout << a.pop() << " is Popped\n";
 cout << a.pop() << " is Popped\n";

 return 0;
}
```

المخرجات OUTPUT:

```
3 is Pushed
10 is Pushed
1 is Pushed

1 is Popped
10 is Popped
3 is Popped
```

يمكننا أن نرى بوضوح أن البيانات التي أدخلت أخيراً قد خرجت من المكدس أولاً. وهذا هو السبب في تسمية المكدس بأنه بنية بيانات LIFO ، أو Last In first Out. أو الداخل أخيراً يخرج أولاً. وأعتقد أنك تعرف لماذا؟

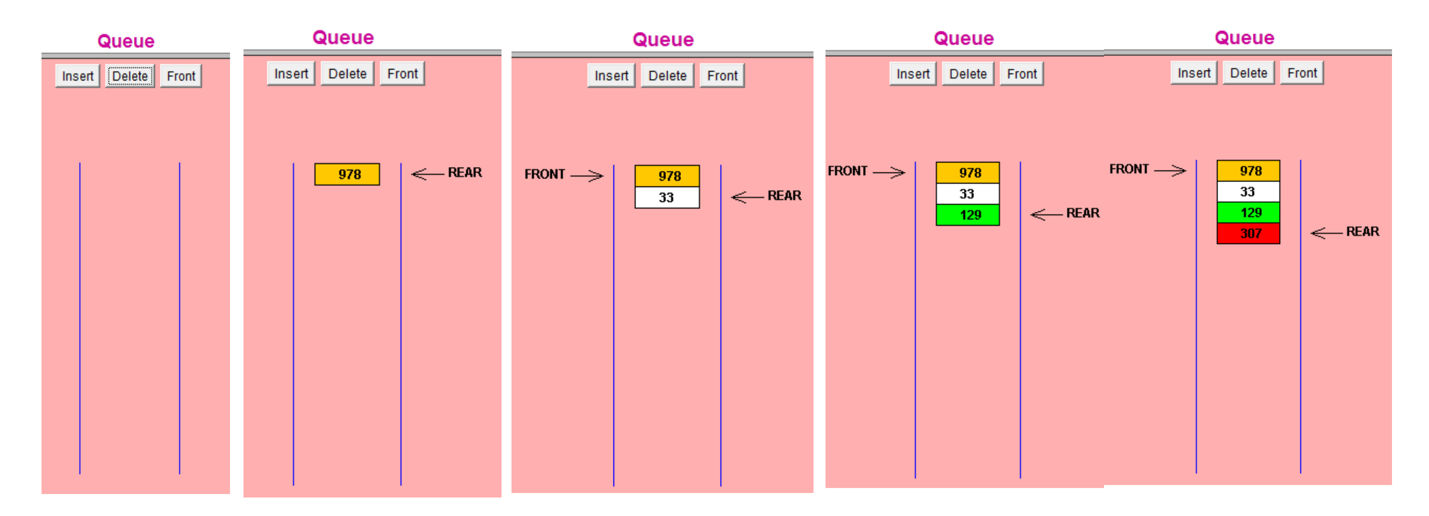
دعونا نرى كيف قمنا بتطبيق المكدس. بدأنا أولاً بإنشاء متغير أسميناه top يشير إلى أعلى المكدس. وقمنا بإعطائه القيمة الابتدائية 1-، وهذا للإشارة إلى أن المكدس فارغ. كلما قمنا بإدخال البيانات، فإن قيمة المتغير top تزداد بمقدار 1، ويتم تخزين البيانات في المصفوفة المسماة arr. يمكن أن نذكر أن هناك عيب واحد لهيكل البيانات هذا. فهنا ذكرنا أن الحد الأقصى لعدد العناصر هو 10. ماذا إذا كنا بحاجة إلى أكثر من 10 عناصر بيانات؟ في هذه الحالة، ولحل هذه المشكلة، يجب أن نقوم بتنفيذ المكدس باستخدام القوائم المتصلة linked lists، وهذا ما سيتم شرحه لاحقاً.

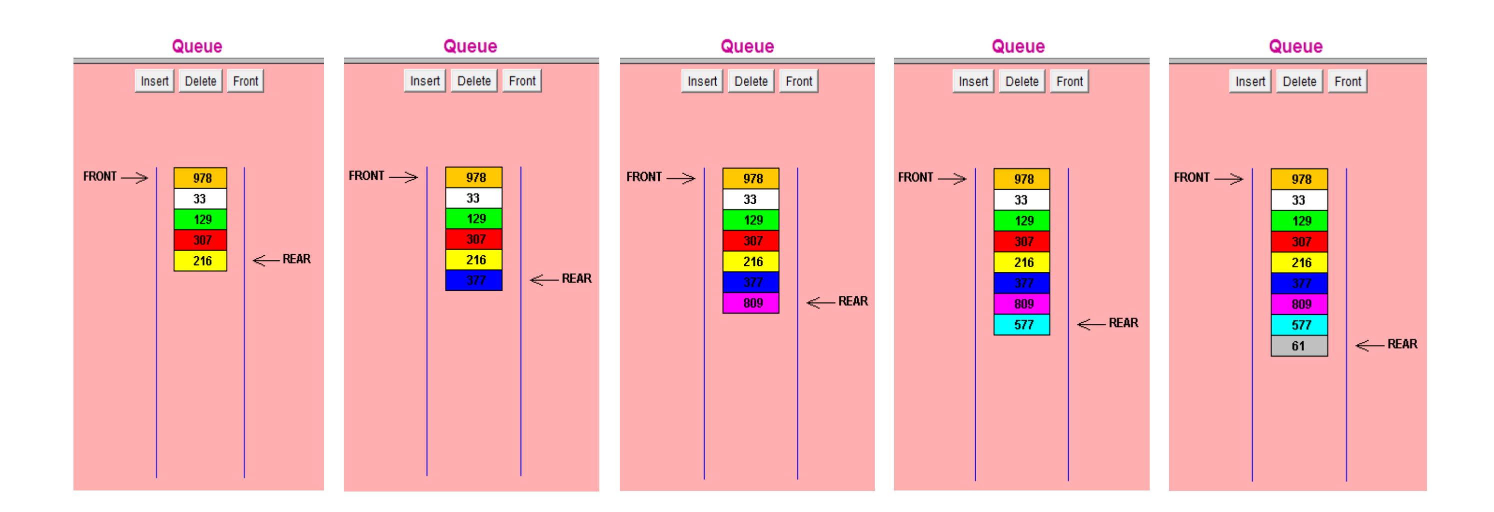
أرجو أن تكون قد استوعبت مفهوم المكدس جيداً. والآن دعونا نتقدم إلى هيكل بياني جديد هو الطابور queue.

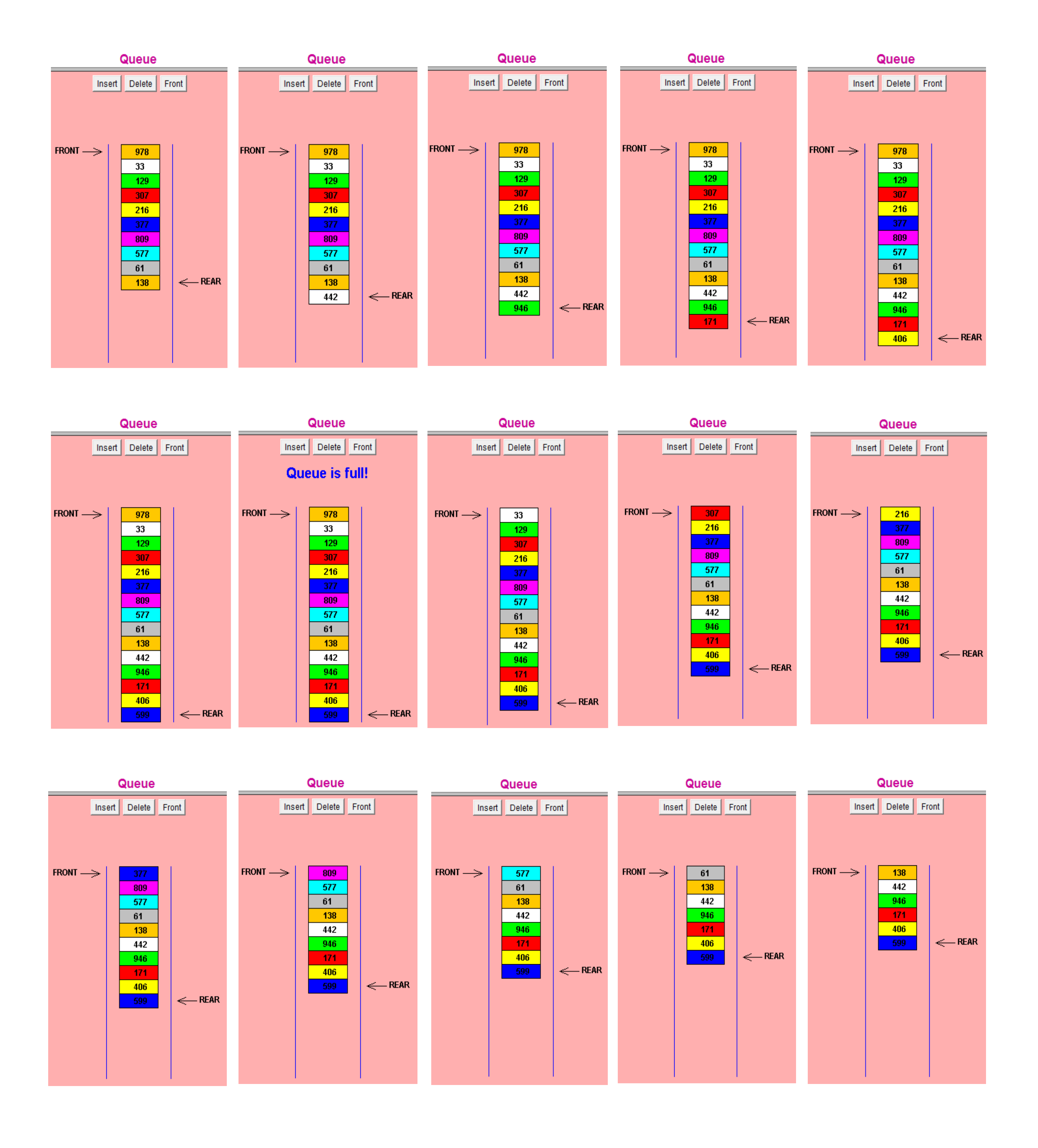
رابعاً: الطابور QUEUE

لنقل إن هناك جمع ضخم من الناس في بقالة خاصة. فهناك الكثير من الناس يحاولون شراء السلع التي يريدونها، وصاحب المتجر لا يعرف من أين يبدأ. فالجميع يريدون إتمام عملية الشراء بسرعة، وصاحب المتجر يحتاج طريقة فعالة لحل هذه المشكلة. ماذا يفعل؟ إن عليه استخدام نظام طابور يقوم على مبدأ "الواصل أولاً، يخدم أولاً" (First Come First Served) أو (First In First Out – FIFO). إن آخر شخص يريد شراء السلع، عليه أن يقف وراء آخر شخص في نهاية END الطابور. لكن نرى أن صاحب المتجر يجلس أمام FRONT طابور الانتظار. وهو يقوم بإعطاء السلع إلى الشخص الذي يتواجد في مقدمة FRONT قائمة الانتظار. وبعد إتمام العملية، على الشخص الموجود في مقدمة FRONT الطابور أن يغادر. وحينها يصبح ثاني شخص في طابور هو الأول في قائمة الانتظار.

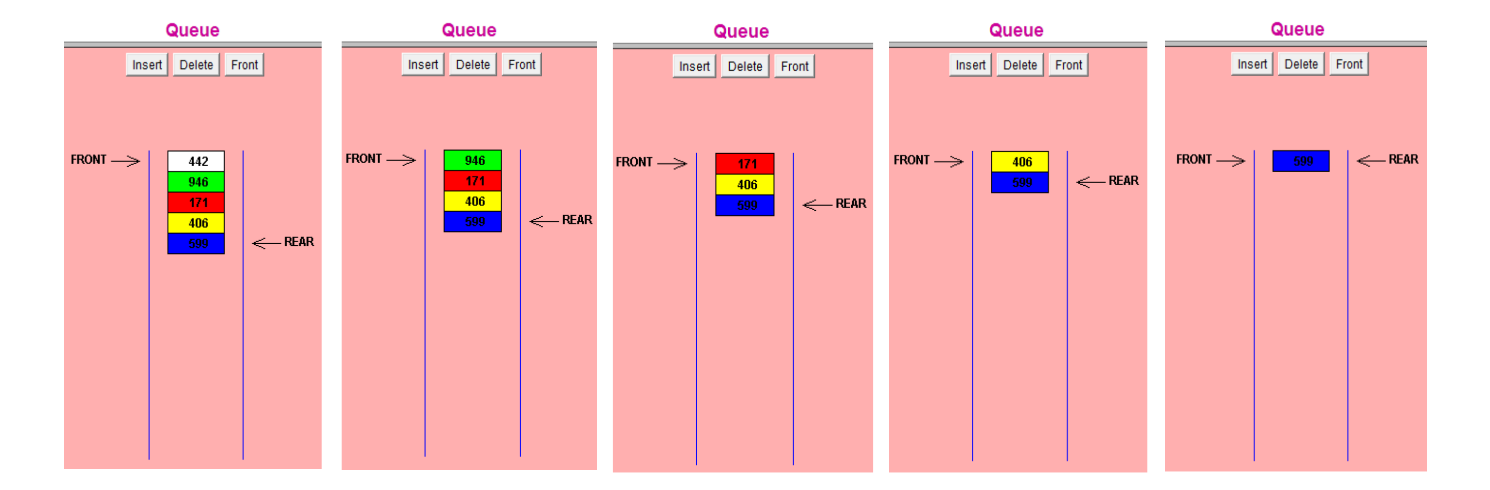
هل استوعبت هذا المفهوم إلى حد الآن؟ إن الطابور Queue يشبه المكدس Stack، إلا أن عملية إضافة البيانات تتم في النهاية الخلفية للطابور، وعملية حذف البيانات تتم من أمام أو مقدمة الطابور. ثم إن عملية كتابة الطابور تعد أصعب بكثير مقارنة بكتابة المكدس. فهنا يجب أن نحتفظ باثنين من المتغيرات الصحيحة عند كتابة هيكل بيانات الطابور، أحدهما يدل على النهاية الأمامية للطابور والآخر يشير إلى النهاية الخلفية للطابور.

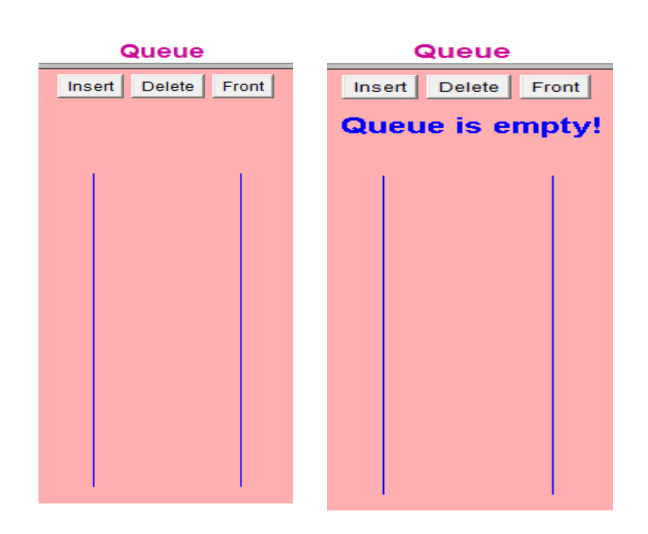






Queue Insert Delete Front FRONT 978 33 129 307 216 377 809 577 61 138 REAR
Queue Insert Delete Front FRONT 978 33 129 307 216 377 809 577 61 138 442 REAR
Queue Insert Delete Front FRONT 978 33 129 307 216 377 809 577 61 138 442 946 REAR
Queue Insert Delete Front FRONT 978 33 129 307 216 377 809 577 61 138 442 946 171 REAR
Queue Insert Delete Front FRONT 978 33 129 307 216 377 809 577 61 138 442 946 171 406 REAR
Queue Insert Delete Front FRONT 978 33 129 307 216 377 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front Queue is full! FRONT 978 33 129 307 216 377 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 33 129 307 216 377 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 307 216 377 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 216 377 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 377 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 809 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 577 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 61 138 442 946 171 406 599 REAR
Queue Insert Delete Front FRONT 138 442 946 171 406 599 REAR





دعونا نستخدم نفس أسلوب التكويد الذي استخدمناه عند إنشاء المكدس. نقوم أولاً بإعطاء قيمة ابتدائية لمؤشري الكلاس كليهما تساوي 1- في إشارة إلى أن الطابور فارغ. عند إضافة البيانات إلى الطابور، فإن كلا الطرفين يحصلان على قيم موجبة. وعند إضافة بيانات جديدة، يتم زيادة مؤشر النهاية الخلفية rear بمقدار واحد، وعندما يتم حذف البيانات يتم إنقاص مؤشر النهاية الأمامية front بمقدار واحد. هذا الأمر يعمل جيداً، لكن مع وجود عيب خطير. ماذا إذا الحد الأقصى للطابور هو 5 عناصر؟. لنفترض أن المستخدم قد أضاف 4 عناصر، ثم حذف 3 عناصر وأضاف عنصرين مرة أخرى. في هذه الحالة لن يسمح الطابور له بإضافة النصف الآخر من البيانات، وسيكون التقرير الذي سيظهر هو أن الطابور ممتلئ. والسبب هو أننا نقوم بالزيادة/النقصان - بطريقة عمياء – اعتماداً على الإضافة / الحذف، غير مدركين أن كلاً من طرفي الطابور مرتبط بالآخر. وسأترك هذا الأمر كتمرين لك لتقوم بحله. لماذا يقوم الطابور هنا بإظهار تقرير يفيد أنه ممتلئ؟، على الرغم من أنه في الواقع ما زال شاغراً؟ هذا يدفعنا إلى القول إننا بحاجة إلى مفهوم آخر للتعامل مع هذا الهيكل البياني. وفي هذا الأسلوب سوف نركز على البيانات أكثر من تركيزنا على نهايتي الإضافة والحذف.

ما نستخدمه الآن هو مثال البقالة مرة أخرى. لنفترض أن هناك 5 عناصر في الطابور، ونحن نريد أن نحذفهم واحداً تلو الآخر. نقوم أولاً بحذف أول عنصر بيانات، وهو العنصر المشار إليه بمؤشر نهاية الحذف. ثم نقوم بإزاحة جميع البيانات خطوة واحدة للأمام، بحيث يصبح العنصر الثاني هو الأول، العنصر والثالث هو الثاني وهكذا .. والطريقة الأخرى هو الاحتفاظ بالفرق بين طرفي الطابور، وهذه طريقة غير عملية. وبالتالي سنتمسك بطريقتنا السابقة. قد تكون هذه الطريقة بطيئة في الطوابير الكبيرة، ولكنها تعمل بشكل جيد ولا شك. فيما يلي سنذكر الكود.

خوارزمية الحذف من بداية الطابور

1. اختبر مؤشر front ، إذا كان يساوي -1 قم بما يلي:
   a. اعرض رسالة "Queue is Empty"
2. وإلا:
   a. كرر ما يلي ابتداءً من j = 0 وانتهاءً بمؤشر rear:
      i. إذا كانت j + 1 أقل من أو تساوي rear
         1. اخزن الموقع j + 1 من الطابور في متغير temp
         2. ضع temp في الموقع j من الطابور
      ii. وإلا قم بما يلي:
         1. انقص مؤشر rear بمقدار 1.
         2. إذا كان مؤشر rear = -1 اجعل مؤشر front = -1
            وإلا اجعل front = 0

خوارزمية الإضافة إلى نهاية الطابور

1. إذا كان front = -1 و rear = -1 قم بما يلي:
   a. زد front بمقدار 1.
   b. زد rear بمقدار 1.
2. وإلا قم بما يلي:
   a. زد rear بمقدار 1.
   b. إذا أصبحت rear مساوية لـ MAX قم بما يلي:
      i. اطبع عبارة "Queue is Full".
      ii. انقص rear بمقدار 1.
      iii. اخرج من الخوارزمية
3. اخزن العنصر المراد إضافته في الموقع rear من الطابور.

```
#include <iostream>

using namespace std;

#define MAX 5            //  أقصى حجم للطابور

class Queue
{
 private:
     int t[MAX];
     int rear;       //  طرف الإضافة
     int front;       //  طرف الحذف

 public:
     Queue()//الباني
     {
          front = -1;
          rear = -1;
     }
```

```
     void del();
     void add(int item);
     void display();
};

void Queue::del()//دالة الحذف من بداية الطابور
     {
          int tmp;
          if(front == -1)
          {
               cout << "Queue is Empty";
          }
          else
          {
               for(int j = 0; j <= rear; j++)
               {
                    if((j+1) <= rear)
                    {
                         tmp = t[j+1];
                         t[j] = tmp;
                    }
                    else
                    {
                         rear--;

                         if(rear == -1)
                              front = -1;
                         else
                              front = 0;
                    }
               }
          }
     }

void Queue::add(int item)//دالة الإضافة إلى نهاية الطابور
     {
          if(front == -1 && rear == -1)
          {
               front++;
               rear++;
          }
          else
          {
               rear++;
               if(rear == MAX)
               {
                    cout << "Queue is Full\n";
                    rear--;
                    return;
               }
          }
          t[rear] = item;
     }
```

```
void Queue::display()//دالة عرض بيانات الطابور
	{
		if(front != -1)
		{
			for(int i = 0 ; i <= rear ; i++)
				cout << t[i] << " ";
		}
		else
			cout << "EMPTY";
	}


int main()
{
	Queue a;//كائن من الكلاس
	int data[5] = {32, 23, 45, 99, 24};

	cout << "Queue before adding Elements: ";
	a.display();
	cout << endl << endl;

	for(int i = 0 ; i < 5 ; i++)
	{
		a.add(data[i]);
		cout << "Addition Number : " << (i+1) << " : ";
		a.display();
		cout << endl;
	}

	cout << endl;
	cout << "Queue after adding Elements: ";
	a.display();
	cout << endl << endl;

	for(i = 0 ; i < 5 ; i++)
	{
		a.del();
		cout << "Deletion Number : " << (i+1) << " : ";
		a.display();
		cout << endl;
	}

	return 0;
}
```

المخرجات OUTPUT:

```
Queue before adding Elements: EMPTY

Addition Number : 1 : 32
Addition Number : 2 : 32 23
Addition Number : 3 : 32 23 45
Addition Number : 4 : 32 23 45 99
Addition Number : 5 : 32 23 45 99 24

Queue after adding Elements: 32 23 45 99 24

Deletion Number : 1 : 23 45 99 24
Deletion Number : 2 : 45 99 24
Deletion Number : 3 : 99 24
Deletion Number : 4 : 24
Deletion Number : 5 : EMPTY
```

وكما يمكنك أن ترى بوضوح من خلال مخرجات هذا البرنامج، فإن الإضافة تتم دائماً في نهاية طابور، في حين يتم الحذف من الطرف الأمامي للطابور. ومرة أخرى سنقوم بتمديد الحد الأقصى من البيانات في وقت لاحق عندما نتعلم موضوع القوائم المتصلة.

خامساً: القوائم المتصلة Linked Lists

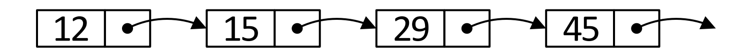
إن القائمة المتصلة linked list هي بنية بيانات أكثر تعقيداً من المكدس والطابور. فالقائمة المتصلة تتكون من قسمين، الأول يمثل النصف الخاص بالبيانات DATA والثاني يمثل النصف الخاص بالمؤشر POINTER. ويحتوي جزء البيانات على البيانات التي نريد تخزينها، في حين أن جزء المؤشر يحتوي على مؤشر يشير إلى العقدة التالية في القائمة المتصلة.

بهذه الطريقة أصبح لدينا بنية بيانات ديناميكية Dynamic Data Structure، حيث يمكننا أن نضيف أي قدر نريده من البيانات بدون قيود على الذاكرة، وهذا ما لم يكن متحققاً في المكدس والطابور سابقاً، واللذان تم تمثيلهما عن طريق المصفوفات، والتي تعد هيكل بياني استاتيكي Static Data Structure (أي أن الحجم الأقصى للهيكل البياني محدد مسبقاً ولا يمكن تجاوزه).

ومن المهم أن نذكر أن المؤشرات pointers تلعب دورا كبيراً في هياكل البيانات ... فعدم وجود المؤشرات، يعني أن لا هياكل البيانات أصلاً... لذا فإن معرفة المعلومات الأساسية عن المؤشرات أمر لا بد منه قبل الاستمرار.

أنظر إلى هذا الرسم البياني الذي يوضح القائمة المتصلة:

العناصر المضافة إلى القائمة المتصلة هي: 45 29 15 12



نلاحظ هنا أن البيانات المخزنة داخل هيكل البيانات هي 45 ,29 ,15 ,12. وكما ترون، فإن مؤشر 12 يشير إلى العقدة التالية في القائمة وهي 15، والتي بدورها تشير إلى 29 وهكذا. والشكل هو عبارة عن مجرد فكرة مفاهيمية عن القوائم المتصلة. إلا أن الحقيقة هي أن تخزين كل هذه البيانات يتم بشكل عشوائي في أماكن عشوائية في الذاكرة. وباستخدام المؤشرات يمكن الحصول على كل البيانات بشكل مرتب كما نريدها.

عند إضافة بيانات إلى قائمة متصلة سنقوم بالتحقق من العقد المضافة مسبقاً. ثم بعد ذلك نصل إلى العقدة الأخيرة من القائمة والتي فيها قيمة المؤشر تكون NULL ونجعلها تشير إلى العقدة الجديدة في القائمة والتي تم إنشاؤها حديثاً. أما إذا لم يكن لدينا عقدة موجودة مسبقاً، فإننا سنقوم ببساطة بإضافة واحدة جديدة ونضع مؤشرها على القيمة NULL.

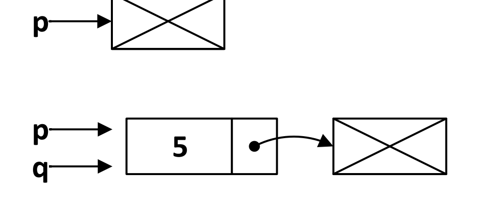
الحذف يعد أكثر تعقيداً. لنفترض أننا نريد حذف العقدة التي فيها القيمة 15. علينا أولاً العثور على القيمة 15. ثم نجعل مؤشر العقدة التي فيها القيمة 12 يشير إلى العقدة التي فيها القيمة 29. ثم بعد ذلك نقوم بحذف العقدة التي تحتوي على القيمة 15.

قم بدراسة الكود المصدري التالي والذي سيساعدك على فهم واستيعاب عمليات القائمة المتصلة:

خوارزمية الاضافة في بداية القائمة المتصلة:

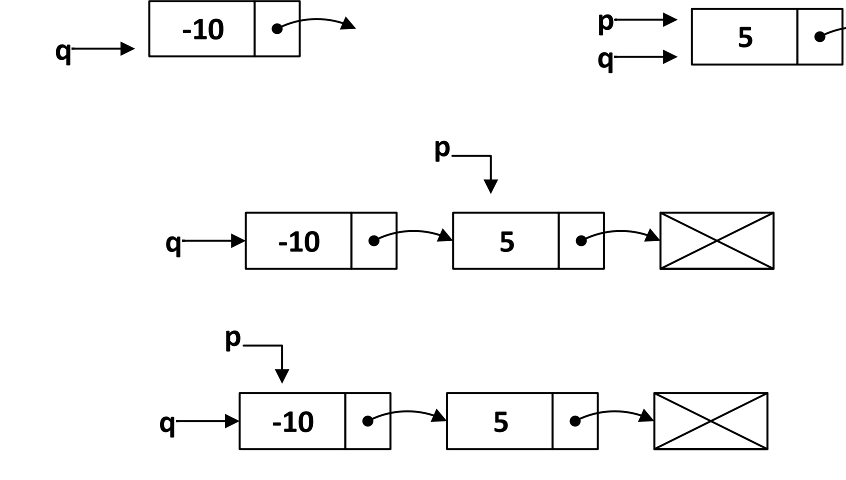
1. أنشئ عقدة q.
2. اجعل جزء بيانات q = العدد المراد إضافته.
3. اجعل جزء مؤشر p = q.
4. اجعل p = q.

مثال : لنفترض أن لدينا قائمة فارغة:



نريد إضافة العدد 5 إلى بداية القائمة:

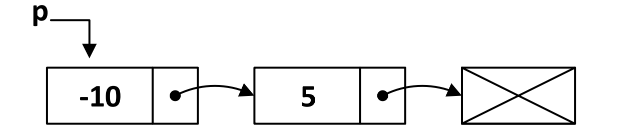
ثم نريد إضافة العدد -10 إلى بداية القائمة:



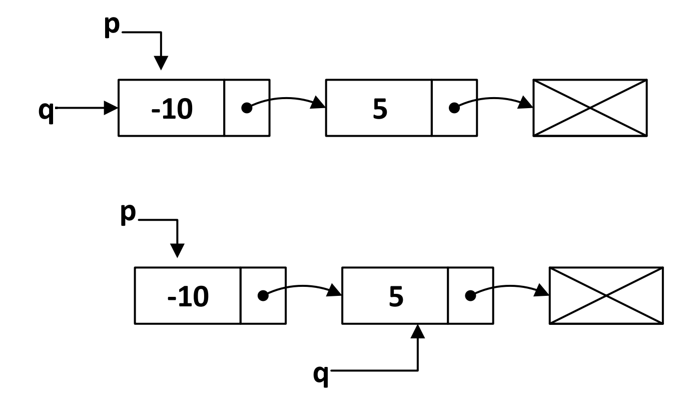
خوارزمية الإضافة في نهاية القائمة المتصلة:

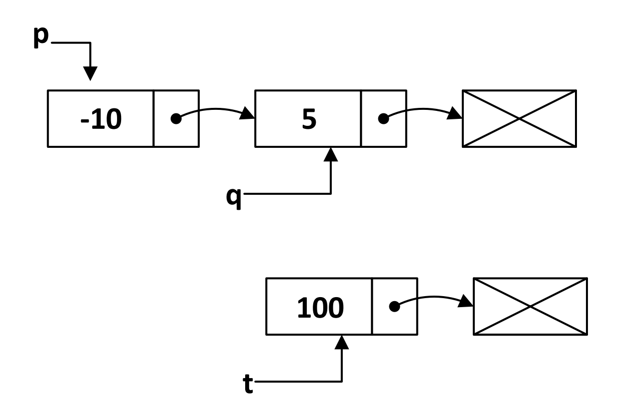
1. **عرف عقدتين (q للانتقال إلى آخر القائمة، و t العقدة التي ستضاف).**
2. **اختبر مؤشر البداية p إذا كان فارغاً قم بمايلي:**
   a. **أنشئ العقدة p.**
   b. **اجعل جزء بيانات p = العدد المراد إضافته.**
   c. **اجعل جزء مؤشر p = NULL.**
3. **وإلا قم بما يلي:**
   a. **اجعل q = p.**
   b. **طالما تالي q ليس فارغاً انقل q إلى العقدة التالية.**
   c. **أنشئ العقدة t.**
   d. **اجعل جزء بيانات t = العدد المراد إضافته.**
   e. **اجعل جزء مؤشر t = NULL.**
   f. **اجعل جزء مؤشر q = t.**

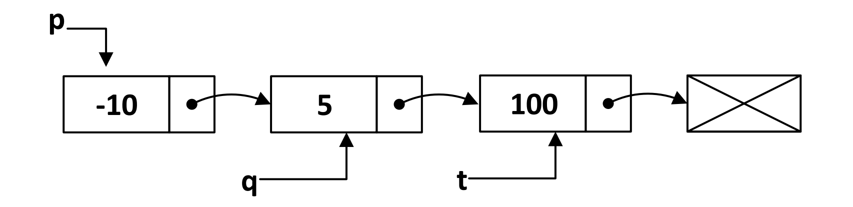
مثال : لنفترض أن لدينا القائمة المتصلة التالية:



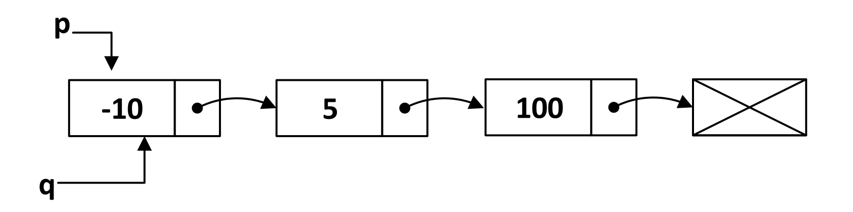
نريد إضافة العدد 100 إلى نهاية هذه القائمة:

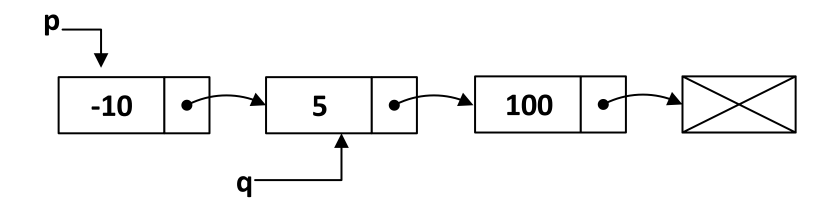


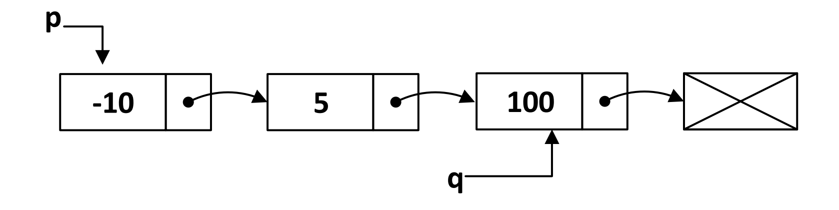


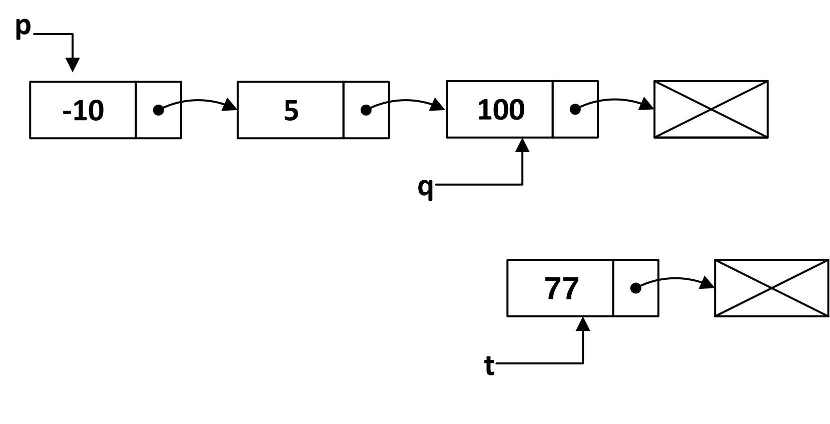


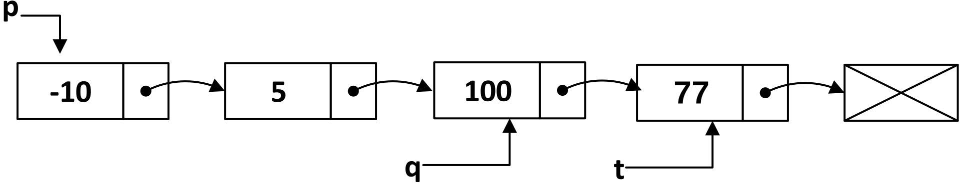
إيضاً إذا أردنا إضافة القيمة 77 **إلى النهاية القائمة السابقة:**







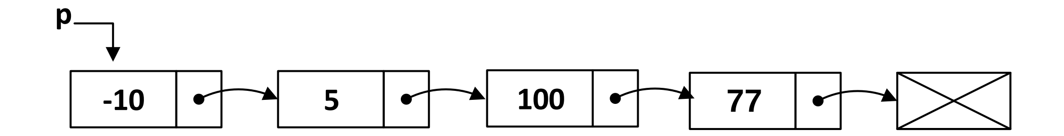




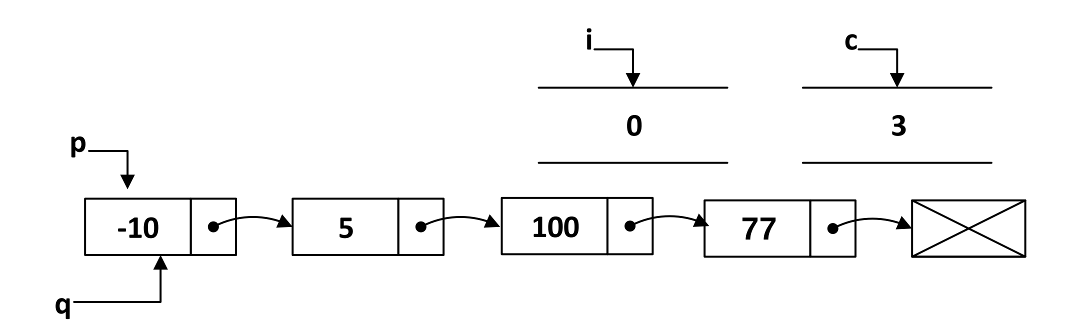
خوارزمية الاضافة بعد موقع معين c:

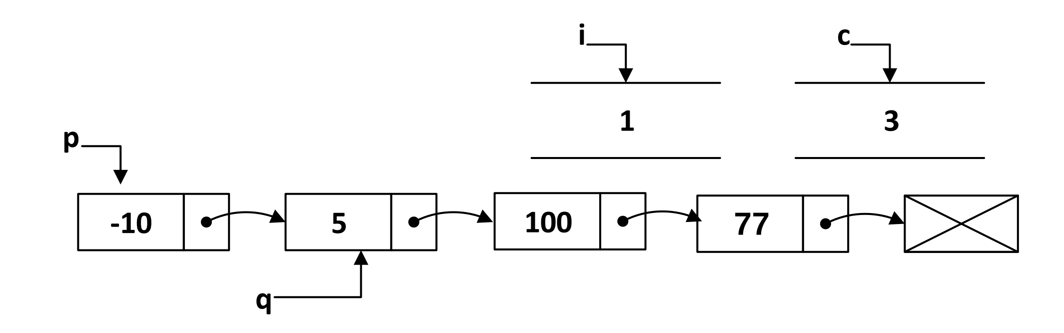
1. عرف عقدتين q و t.
2. عرف i = 0.
3. اجعل q = p.
4. طالما i أقل من c (الموقع المراد الإضافة بعده) قم بما يلي:
   a. حرك q إلى العقدة التالية.
   b. إذا كانت q تساوي NULL قم بما يلي:
      i. اعرض رسالة "عدد العناصر أقل من قيمة الموقع المراد الإضافة بعده"
      ii. أخرج من الخوارزمية.
   c. زد قيمة i بمقدار واحد.
5. أنشئ العقدة t.
6. اجعل جزء بيانات t = العدد المراد إضافته.
7. اجعل جزء مؤشر t = جزء مؤشر q.
8. اجعل جزء مؤشر q = t.

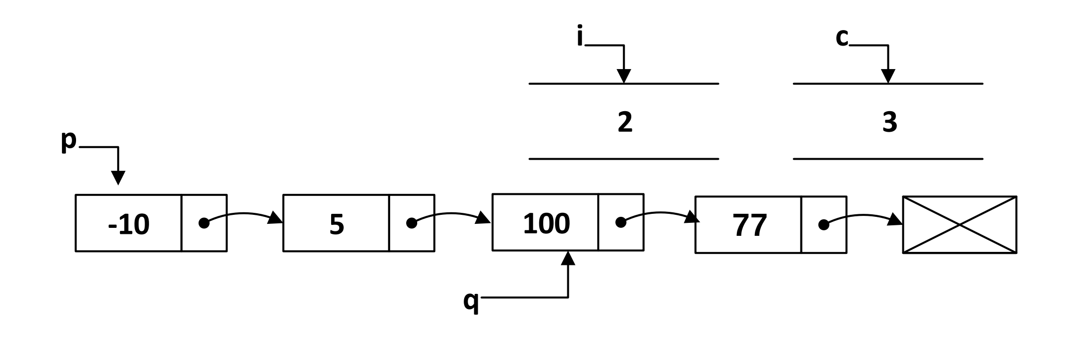
مثال : لدينا القائمة التالية.

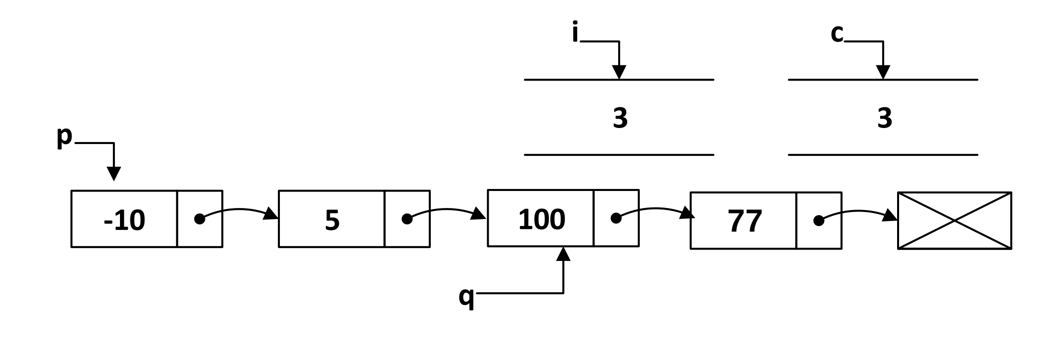


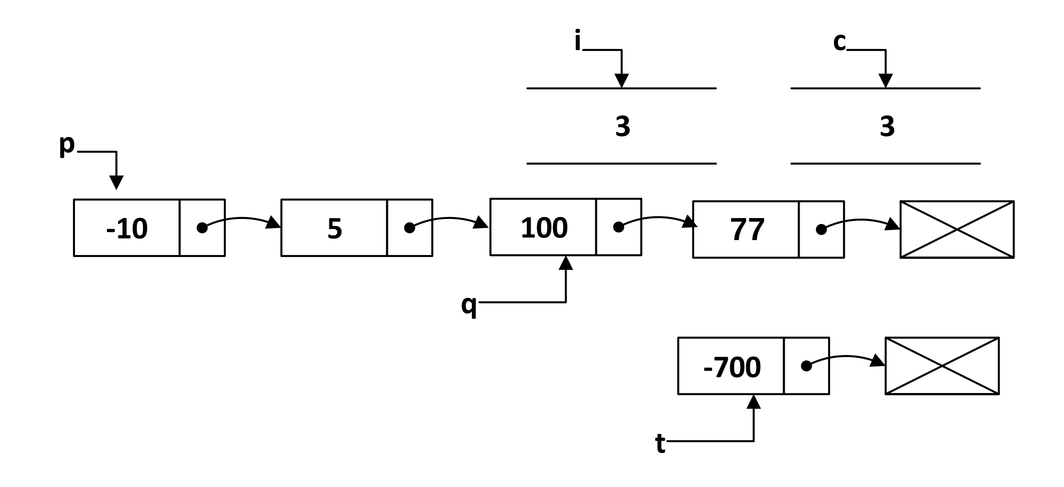
نريد إضافة القيمة 700- بعد الموقع 3.

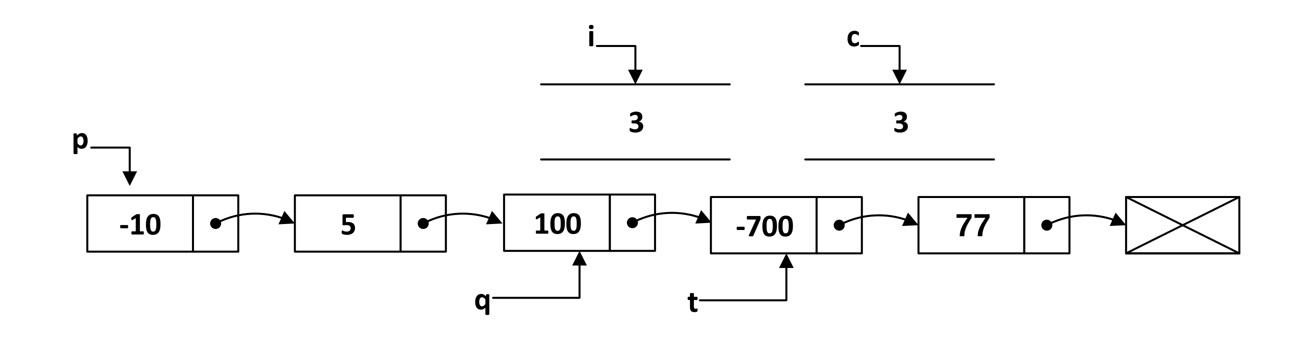





i
c
2
3
p
-10
5
100
77
q


i
c
3
3
p
-10
5
100
77
q

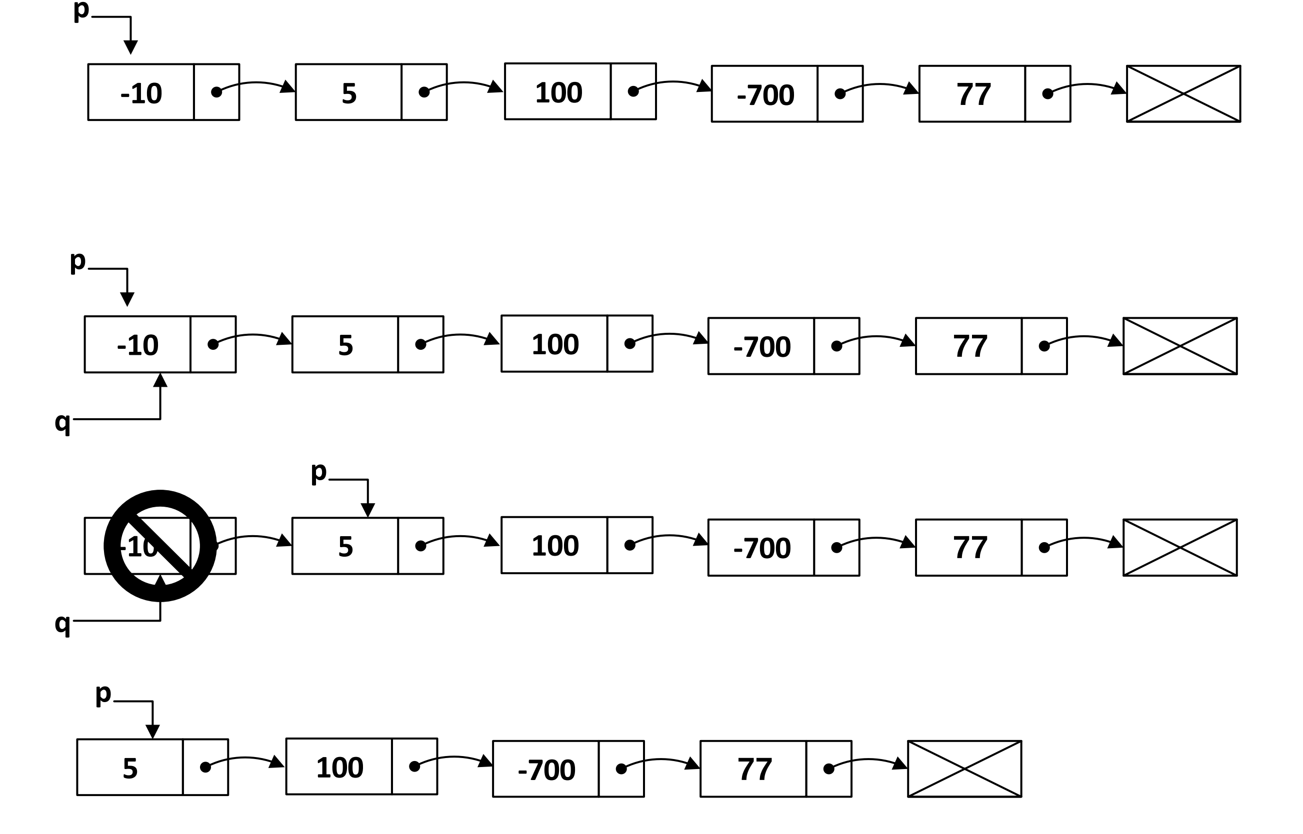

i
c
3
3
p
-10
5
100
77
q
-700
t


i
c
3
3
p
-10
5
100
-700
77
q
t

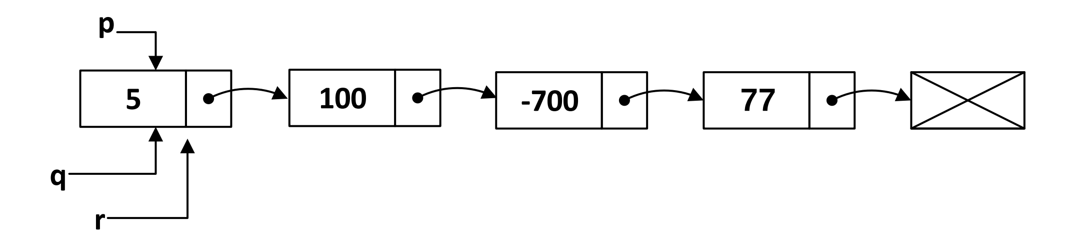
خوارزمية حذف قيمة من القائمة المتصلة:

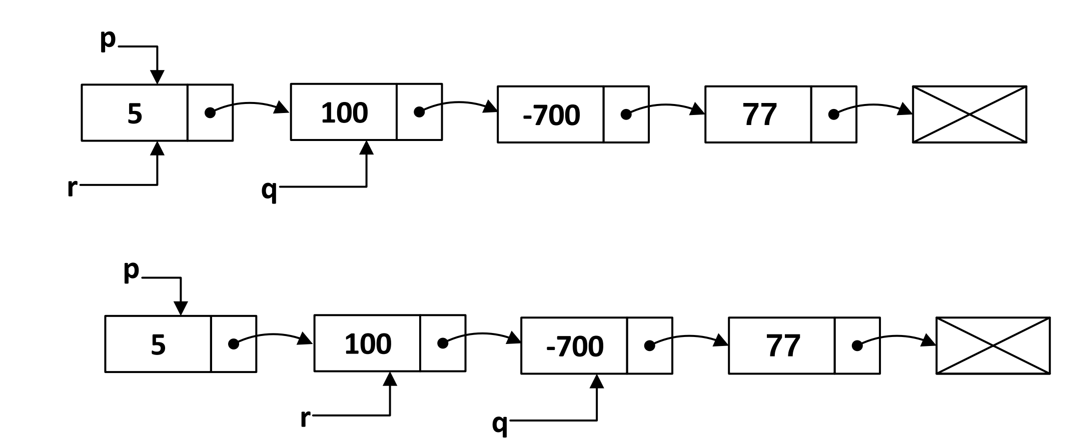
1. عرف عقدتين q و r.
2. اجعل q = p.
3. إذاكان جزء بيانات q يساوي العدد المراد حذفه ، قم بما يلي:
    a. اجعل p يساوي تالي q.
    b. احذف العقدة q.
    c. اخرج من الخوارزمية.
4. اجعل r = q.
5. طالما q لا تساوي NULL قم بما يلي:
    a. إذا كان جزء بيانات q يساوي العدد المراد حذفه، قم بما يلي:
        i. اجعل تالي r مساوياً لتالي q.
        ii. احذف العقدة q.
        iii. اخرج من الخوارزمية.
    b. اجعل r = q.
    c. حرك q إلى العقدة التالية.
6. اطبع عبارة "العدد المراد حذفه غير موجود".

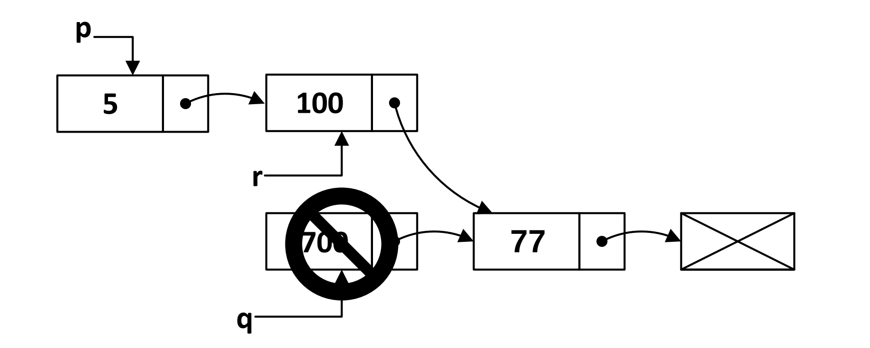
مثال : لدينا القائمة المتصلة التالية، ونريد حذف العنصر 10-:

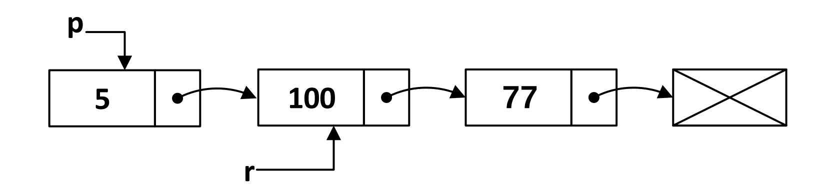


أما إذا أردنا حذف العنصر 700- في القائمة الأخيرة أعلاه، فسنقوم بما يلي:

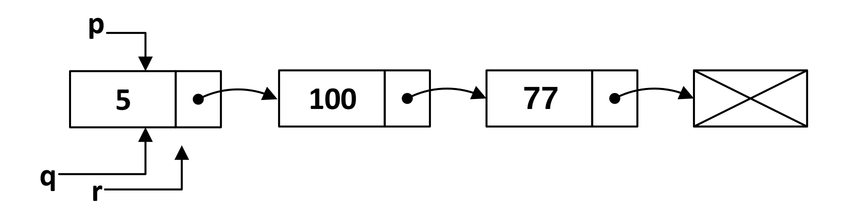


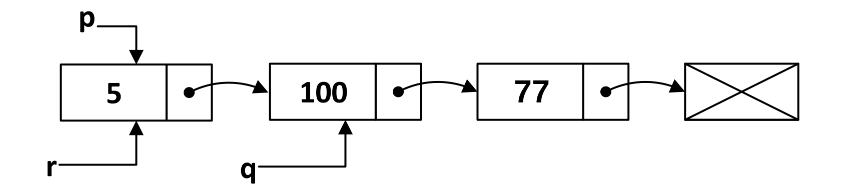


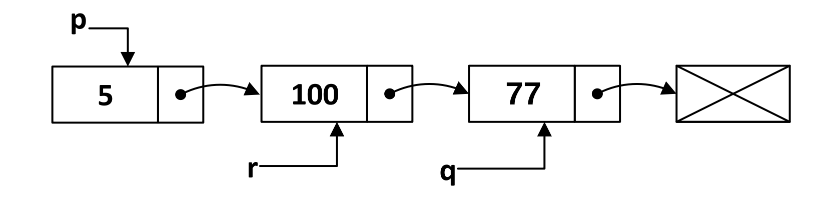


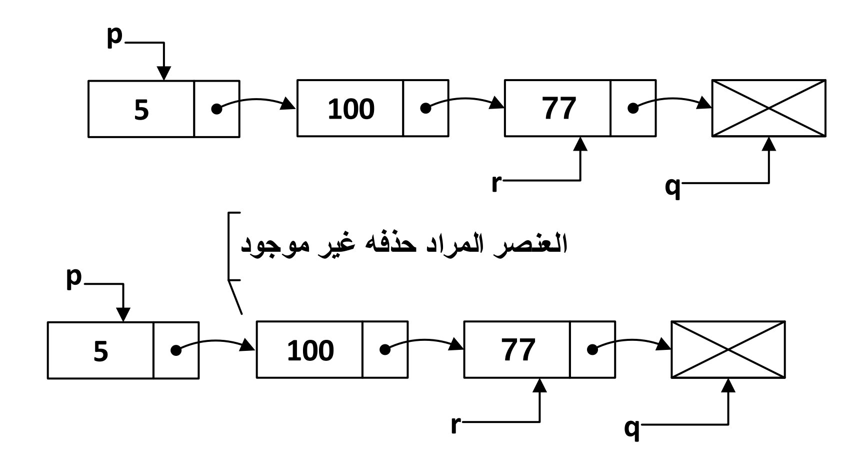


إما إذا كان العنصر المراد حذفه في القائمة الأخير أعلاه هو 200 فسيتم ما يلي:










```
#include <iostream>
using namespace std;

struct node //تعريف العقدة
{
	int data;
	node *next;
};

class LinkList
{
	private:
		node *p;
	public:
		LinkList();
		void add_as_first( int num );
		void append( int num );
		void addafter( int c, int num );
		void del( int num );
		void display();
		int count();
		~LinkList();
};
```

```
LinkList::LinkList()//الباني
{
     p = NULL;
}

void LinkList::add_as_first(int num)//دالة الإضافة في البداية
{
   node *q;

   q = new node;
   q -> data = num;
   q -> next = p;
   p = q;
}

void LinkList::append(int num)//دالة الإضافة في النهاية
{
   node *q,*t;

   if( p == NULL )
   {
      p = new node;
      p -> data = num;
      p -> next = NULL;
   }
   else
   {
      q = p;
      while(q -> next != NULL )
          q = q -> next;

      t = new node;
      t -> data = num;
      t -> next = NULL;
      q -> next = t;
   }
}

void LinkList::addafter( int c, int num)//دالة الإضافة بعد موقع معين
{
   node *q, *t;
   int i;
   for(i = 0, q = p; i < c; i++)
   {
      q = q -> next;
      if( q == NULL )
      {
          cout << "\nThere are less than " << c << " elements.";
          return;
      }
   }
```

```
   t = new node;
   t -> data = num;
   t -> next = q -> next;
   q -> next = t;
}

void LinkList::del( int num )//دالة الحذف
{
   node *q,*r;
   q = p;
   if(q -> data == num )
   {
      p = q -> next;
      delete q;
      return;
   }

   r = q;
   while( q != NULL )
   {
      if( q -> data == num )
      {
          r -> next = q -> next;
          delete q;
          return;
      }

      r = q;
      q = q -> next;
   }
   cout << "\nElement " << num << " not Found.";
}

void LinkList::display()//دالة عرض بيانات القائمة المتصلة
{
   node *q;
   cout << endl;

   for(q = p ; q != NULL ; q = q -> next )
     cout << endl << q -> data;

}
```

```
int LinkList::count()//دالة عد عناصر القائمة المتصلة
{
   node *q;
   int c = 0;
   for(q = p; q != NULL; q = q -> next )
     c++;

   return c;
}

LinkList::~LinkList()//الهادم
{
   node *q;
   if( p == NULL )
     return;

   while( p != NULL )
   {
      q = p -> next;
      delete p;
      p = q;
   }
}

int main()
{
   LinkList ll;
   cout << "No. of elements = " << ll.count();
   ll.append(12);
   ll.append(13);
   ll.append(23);
   ll.append(43);
   ll.append(44);
   ll.append(50);

   ll.add_as_first(2);
   ll.add_as_first(1);

   ll.addafter(3,333);
   ll.addafter(6,666);

   ll.display();
   cout << "\nNo. of elements = " << ll.count();

   ll.del(333);
   ll.del(12);
   ll.del(98);
   cout << "\nNo. of elements = " << ll.count();
   return 0;
}
```

المخرجات OUTPUT:

```
No. of elements= 0
1
2
12
13
333
23
43
666
44
50
No. of elements= 10
Element 98 not found.
No. of elements = 8
```

وهنا كما ترون، فإن الـ class يحتوي على عقدة هيكلية تتكون من قيمة ذات نوع بياني صحيح، ومؤشر يشير إلى العقدة الهيكلية التالية. ونحن هنا نحتفظ بعقدة مؤشرة p تشير دائماً إلى العنصر الأول في القائمة. وفيما يلي سرد لقائمة الدوال التي تم استخدامها في هيكل البيانات هذا.

```
LinkList();                              //الباني
void append( int num );                  //الإضافة في نهاية القائمة
void add_as_first( int num );            //الإضافة في بداية القائمة
void addafter( int c, int num );         //إضافة البيانات بعد موقع محدد
void del( int num );                     //حذف بيانات محددة
void display();                          //عرض بيانات القائمة المتصلة
int count();                             //عدد العناصر في القائمة المتصلة
~LinkList();                             //الهادم
```

سترى في أماكن عديدة جملاً برمجية مثل `q = q -> next` داخل حلقة. هذا الجملة تقوم فقط بتحريك المؤشر من عقدة إلى أخرى. ويقوم الهادم destructor، وكذلك الدالة del باستخدام عامل الحذف delete لإعادة تخصيص المساحة التخزينية التي تم تخصيصها مسبقاً باستخدام عامل الحجز new. بقية الكود ستكون واضحة إذا كان لديك فهم أساسيات المؤشرات.

إن من مزايا استخدام المؤشرات هو أنك لن تقلق فيما يخص إهدار المساحة التخزينية من خلال تخصيص الكثير من الذاكرة بشكل مسبق. فكلما كانت هناك حاجة إلى زيادة البيانات، يتم تخصيص الذاكرة وفقاً لذلك. لكن الجانب الآخر هو أنه لكي نصل إلى كل عقدة فعلينا أن نمر من خلال كل العقد حتى نصل إلى العقدة المطلوبة. وهذا هو السبب في وجود أشكال مختلفة للقوائم المتصلة من أجل تسهيل الوصول. على سبيل المثال، القوائم المتصلة الدائرية circular linked lists والقوائم المتصلة المضاعفة doubly linked lists. فالقوائم المتصلة الدائرية هي تلك التي تشير فيها آخر عقدة بشكل دائم إلى أول عقدة. والقوائم المتصلة المضاعفة تحتوي على مؤشرين، أحدهما يشير إلى العقدة التالية، والآخر يشير إلى العقدة السابقة.

وسأعطيكم فقط الشفرة المصدرية للقائمة المتصلة الدائرية، في حين كود القائمة المتصلة المضاعفة سيكون تمرين يجب عليكم حله. ومع ذلك، إذا لم تستطع كتابته، فلديك مطلق الحرية في التواصل معي على بريدي الإلكتروني.

سادساً: المكدسات باستخدام القوائم المتصلة STACKS USING LINKED LISTS

سنقوم هنا باستخدام نفس مفهوم المكدس ولكن بدون استخدام قيد الحد الأقصى للبيانات MAXIMUM. وبما أننا سنستخدم القوائم المتصلة لتخزين البيانات في المكدس، فإن المكدس يمكنه أن يحمل أي قدر من البيانات يريده طالما أن البيانات لم تتجاوز حدود الذاكرة. وفيما يلي الشفرة المصدرية:

```
#include <iostream>

using namespace std;

struct node//تعريف العقدة
{
   int data;
   node *next;
};

class LLStack
{
      private:
              node* top;

      public:
             LLStack()//الباني
             {
               top=NULL;
             }

             void push(int n)//الإدخال إلى المكدس
             {
               node *tmp;
               tmp = new node;
               if(tmp == NULL)
                  cout << "\nSTACK FULL";

               tmp -> data=n;
               tmp -> next=top;
               top = tmp;
             }

             int pop()//الإخراج من المكدس
             {
               if(top == NULL)
               {
                    Cout << "\nSTACK EMPTY";
                    return NULL;
               }
                node *tmp;
                int n;
                tmp = top;
```

```
                n = tmp -> data;
                top = top -> next;
                delete tmp;
                return n;
            }

            ~LLStack()//الهادم
            {
              if(top==NULL)
                 return;

              node *tmp;
              while(top != NULL)
              {
                 tmp = top;
                 top = top -> next;
                 delete tmp;
              }
            }
};

int main()
{
     LLStack s;
     s.push(11);
     s.push(101);
     s.push(99);
     s.push(78);
     cout << "Item Popped = " << s.pop() << endl;
     cout << "Item Popped = " << s.pop() << endl;
     cout << "Item Popped = " << s.pop() << endl;
     return 0;
}
```

سابعاً: الطوابير باستخدام القوائم المتصلة QUEUES USING LINKED LIST

بشكل مماثل لما قلناه أعلاه بخصوص المكدس باستخدام القوائم المتصلة، فإن تنفيذ الطابور باستخدام القوائم المتصلة يجعلنا نستغني أيضاً عن قيد الحد الأقصى للبيانات. وفيما يلي الشفرة المصدرية:

```
#include <iostream>

using namespace std;

struct node//تعريف العقدة
{
   int data;
   node *next;
};
```

```
class LLQueue
{
 private:
    node *front, *rear;

 public:
    LLQueue()//الباني
    {
     front = NULL;
     rear = NULL;
    }


    void add(int n)//الإضافة في نهاية الطابور
    {
     node *tmp;
     tmp = new node;
     if(tmp == NULL)
        cout << "\nQUEUE FULL";

     tmp -> data = n;
     tmp -> next = NULL;
     if(front == NULL)
     {
          rear = front = tmp;
          return;
     }
            rear->next=tmp;
            rear=rear->next;
    }


    int del()//الحذف من بداية الطابور
    {
     if(front == NULL)
     {
          Cout << "\nQUEUE EMPTY";
          return NULL;
     }
        node *tmp;
        int n;
        n = front -> data;
        tmp = front;
        front = front -> next;
        delete tmp;
        return n;
    }
```

```
    ~LLQueue()//الهادم
    {
          if(front == NULL)
             return;
          node *tmp;
          while(front != NULL)
          {
               tmp = front;
               front = front -> next;
               delete tmp;
          }
     }
};

int main()
{
     LLQueue q;
     q.add(11);
     q.add(22);
     q.add(33);
     q.add(44);
     q.add(55);
     cout << "\nItem Deleted = " << q.del();
     cout << "\nItem Deleted = " << q.del();
     cout << "\nItem Deleted = " << q.del();
     return 0;
}
```

ثامناً: القوائم المتصلة الدائرية CIRCULAR LINKED LISTS

إن القوائم المتصلة الدائرية هي تماماً مثل القوائم المتصلة الطبيعية مع فارق أن المؤشر في العقدة الأخيرة في القائمة يشير إلى العقدة الأولى في القائمة. وقد تتساءل هنا ... لماذا قد نريد أن نفعل مثل هذا الشيء؟ حسنا... هل تعلم أن القوائم المتصلة الدائرية تستخدم تقريباً في كثير من المواقف، فهي مثلاً تستخدم في الإعلانات الالكترونية حيث يتم إضافة كل إعلان إلى القائمة ويتم بعد ذلك عرضه. وبعد عر آخر إعلان سوف يتم تلقائياً عرض الإعلان الأول في القائمة.

والآن، دعونا نرى كيف يمكننا تطبيق القائمة المتصلة الدائرية. لقد كتبت هذه الشفرة بمزيد من التفاصيل بالإضافة إلى أني قمت بتضمين ميزة عرض الشرائح التي تظهر البيانات في قائمة بعد مرور فترة من الوقت. ويستمر عرض البيانات حتى يتم الضغط على أي مفتاح. دعونا نأخذ نظرة:

* ملاحظة: *

إذا لم تكن تستخدم Windows، اتبع الخطوات التالية:

قم بإزالة `#include <windows.h>`

قم بإزالة `#include <conio.h>`

قم بإزالة دالة عرض الشرائح `slideshow()` ودالة الانتظار `wait()` من الكلاس `CirLinkedList`.

قم بإزالة استدعاء دالة عرض الشرائح `slideshow()` في الدالة الرئيسية `main()`.

```
#include <windows.h>
#include <iostream>
#include <conio.h>

using namespace std;

class CirLinkedList
{
   private:
      struct node//تعريف العقدة
      {
          int data;
          node *next;
      };

     node *p;

     public:
          CirLinkedList();//باني افتراضي
          CirLinkedList(CirLinkedList& l);//باني يستقبل قيمة
          ~CirLinkedList();//الهادم
          void add(int);//الإضافة في النهاية
          void del();//الحذف
          void addatbeg(int);//الإضافة في البداية
          void display();//عرض البيانات
          void slideshow(float,int,int);//عرض الشرائح
          int count();//عد العناصر
          void wait(float);//الانتظار
          bool operator ==(CirLinkedList);
          bool operator !=(CirLinkedList);
          void operator =(CirLinkedList);
};

   CirLinkedList::CirLinkedList()
   {
      p=NULL;
   }

   CirLinkedList::CirLinkedList(CirLinkedList& l)
   {
      node *x;
      p = NULL;
      x = l.p;
      if(x == NULL)
          return;
```

```
    for(int i = 1;i <= l.count(); i++)
    {
        add(x -> data);
        x = x -> next;
    }
}

CirNextedList::~CirNextedList()
{
  node *q,*t;
    q = p;
    t = p;
    if(p == NULL)
        return;

    while(q -> next != t)
    {
        p = q;
        q = q -> next;
        delete p;
    }
    p = q;
    delete p;
}

void CirNextedList::add(int n)
{
    if(p == NULL)
    {
        node *q;
        q = new node;
        q -> data = n;
        q -> next = q;
        p = q;
        return;
    }
    node *q;
    q = p;
    while(q -> next != p)
        q = q -> next;

    node *t;
    t = new node;
    t -> data = n;
    t -> next = p;
    q -> next = t;
}

void CirNextedList::display()
{
    if(p == NULL)
    {
        cout << "EMPTY LIST\n";
```

```
        return;
    }
    node *q;
    q = p;
    for(int i = 1; i <= this -> count(); i++)
    {
        cout << q -> data << endl;
        q = q -> next;
    }
}

int CirNextedList::count()
{
    node *q;
    q = p;
    int c = 0;
    if(p == NULL)
        return 0;
    else
        c++;
    while(q -> next != p)
    {
       c++;
       q = q -> next;
    }
    return c;
}

void CirNextedList::del()
{
    if(p == NULL)
        return;

    if(p -> next == p)
    {
        p = NULL;
    }
    else
    {
         node *q;
         q = p;
         while(q -> next != p )
              q = q -> next;

         q -> next = p -> next;
         q = p;
         p = (q -> next == NULL ? NULL : p -> next);
         delete q;
    }
}
```

```
void CirNextedList::addatbeg(int n)
{
   node *q,*t;
   q = p;
   while(q -> next != p)
       q = q -> next;

   t = new node;
   t -> data = n;
   t -> next = p;
   q -> next = t;
   p = t;
}

void CirNextedList::slideshow(float dlay,int x,int y)
{
 /*  if(p == NULL)
   {
       gotoxy(x,y);
       cout << "EMPTY LIST\n";
       return;
   }
   node *q;
   q = p;
   while(!kbhit())
   {
       gotoxy(x,y);
       cout << "               ";
       gotoxy(x,y);
       cout << q -> data;
       wait(dlay);
       q = q -> next;
   }*/
}

void CirNextedList::wait(float t)
{
   long time = GetTickCount()+(t*1000L);
   while(GetTickCount() <= time)
   {
   /*        WAIT !!! */
   }
}

bool CirNextedList::operator ==(CirNextedList t)
{
   if(t.p == NULL && p == NULL)
       return 1;

   if(this -> count() != t.count())
       return 0;

   node *q;
```

```
        q = p;
        bool flag;
        flag = 1;
        node *a;
        a = t.p;
        for(int i = 1; i <= count(); i++)
        {
            if(a -> data != q -> data)
                 flag = 0;

            a = a -> next;
            q = q -> next;
        }
        if(a -> data != q -> data)
            flag = 0;
        return flag;
    }

    bool CirNextedList::operator !=(CirNextedList t)
    {
      return !(this -> operator == (t));
    }



    int main()
    {

        CirNextedList a;
        a.add(1);
        a.add(2);
        a.add(3);
        a.add(4);
        a.addatbeg(128);
        a.del(); // 128 is deleted
        cout<<"\nLIST DATA:\n";
        a.display();

        CirNextedList b=a;
        if(b!=a)
            cout<<endl<<"NOT EQUAL"<<endl;
        else
            cout<<endl<<"EQUAL"<<endl;
        a.slideshow(1,13,13);

        return 0;
}
```

وهنا مرة أخرى تأكدنا من أن العقدة الأخيرة تشير دائماً إلى العقدة الأولى. كل شيء يبدو جيداً. وينبغي أن تكون التعليقات كافية لشرح الشفرة. إن الجزء المثير للاهتمام في هذه الشفرة هو دالة عرض الشرائح `slideshow()`. فهي بوضوح تعرض القائمة في حلقة لا نهائية يمكن إنهاؤها بالضغط على أي مفتاح. ودالة الانتظار `wait()` تسمح بتأخير زمني معين إلى أن يتم ضغط مفتاح عن طريق الدالة `kbhit()`.

والآن نأتي إلى الاختبار. نريد عمل قائمة متصلة مضاعفة (أي يمكن التنقل فيها بالجهتين)، قم بكتابة قائمة متصلة مشابهة فقط مع التغييرات التالية:

1) يجب أن تكون العقدة الهيكلية على هذا النحو:

```
struct node
        {
          int data;
          node *next;  //  مؤشر إلى العقدة التالية
          node *prev;  //  مؤشر إلى العقدة السابقة
        };
```

2) تذكر أنه عند أي عملية إضافة أو حذف فإن مؤشر التالي والسابق يجب أن تتغير وفقاً لذلك.

3) قم بتضمين دالة عرض display تستقبل وسيطاً كما يلي:

```
void NextList::display(int type)
       {
            if(type==1)
            {
             //  شفرة الإخراج من العقدة الأولى إلى العقدة الأخيرة
            }
            else
            {
             //  شفرة الإخراج من العقدة الأخيرة إلى العقدة الأولى
            }
       }
```

هذه الدالة في الحقيقة سهلة الكتابة إذا فهمت كيفية استخدام مؤشري السابق والتالي. إذا وجدت مشاكل بهذا الخصوص راسلني على بريدي الإلكتروني.

تاسعاً: أشجار البحث الثنائي BINARY SEARCH TREES

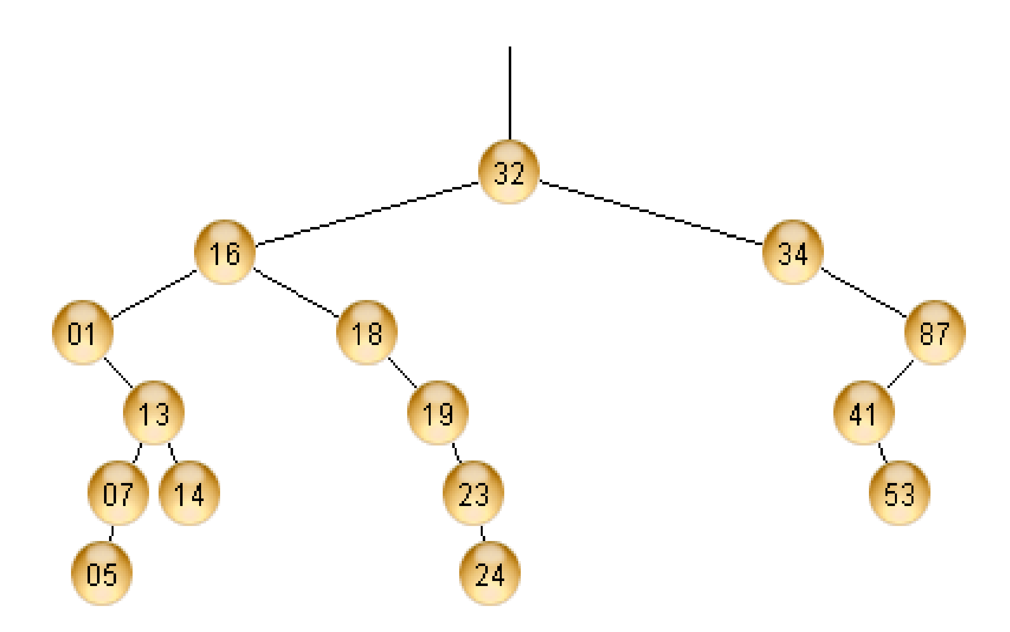
لحد الآن فإن جميع هياكل البيانات التي قمنا بعرضها (المكدس، الطابور، القائمة المتصلة) تمتلك طبيعة خطية، بمعنى أنها تمتلك طريقة واضحة في ترتيب البيانات. والآن نحن بصدد دراسة الأشجار الثنائية BINARY TREES والتي تتطلب عملية تفكير مختلفة لأنه هيكل بيانات غير خطي.

تتكون الشجرة الثنائية من العقدة الرئيسية main node المعروفة باسم الجذر root. والجذر بدوره يمتلك جزئين فرعيين، أي النصف الأيسر left half والنصف الأيمن right half. ثم إن البيانات المخزنة بعد ذلك يتم إنشاؤها اعتماداً على مقارنتها بقيمة الجذر. لنفترض أن قيمة الجذر هي (10) والقيمة التي نريد إضافتها هي 15، فعندها سيتم إضافة البيانات إلى القسم الأيمن من الجذر.

والفكرة الأساسية هي أنه يمكن أن ننظر إلى كل عقدة على أنها شجرة ثنائية مستقلة بذاتها. فلكل عقدة اثنين من المؤشرات، واحد إلى اليسار وآخر إلى اليمين. واعتماداً على القيمة التي سيتم خزنها، يتم وضعها بعد مؤشر العقدة اليمنى إذا كانت قيمة العقدة أقل من القيمة المراد إضافتها، والعكس فإنه يتم وضعها بعد مؤشر العقدة اليسرى إذا كانت قيمة العقدة أكبر من القيمة المراد إضافتها.

دعونا نأخذ مثالاً. لإضافة قائمة الأرقام التالية من اليمين إلى الشمال، سنحصل في نهاية المطاف على شجرة ثنائية كما في الشكل التالي:

53 5 41 24 23 19 14 18 7 13 87 1 34 16 32



واليك الطريقة:

**: احفظ عملية إضافة البيانات إلى الشجرة على ورقة بعد كل خطوة من أجل فهم كيف ستتكون الشجرة.

1. بما أن 32 هو الرقم الأول المراد إضافته، فستكون 32 هي جذر الشجرة.
2. الرقم التالي هو 16 وهو أقل من 32 وعليه ستكون 16 هي العقدة اليسرى لـ 32.
3. فيما يخص الرقم 34. فبما أن 34 أكبر من 32 فإن 34 ستصبح العقدة اليمنى للجذر.
4. الرقم 1. بما أن 1 أقل من 32 سنقفز إلى العقدة اليسارية للجذر. وبما أن العقدة اليسارية قد تم بالفعل أخذها مسبقاً، سنقوم باختبار الرقم 1 مرة أخرى. بما أن 1 أقل من 16، فسيكون هو العقدة اليسارية للعقدة 16.
5. الرقم87. بما أن 87 أكبر من 32، سنقفز إلى العقدة اليمينية للجذر. ومرة أخرى، نجد أنها محجوزة مسبقاً بالقيمة 34. والآن بما أن 87 أكبر من 34، فستكون الـ 87 هي العقدة اليمينية لـ 34.
6. الرقم 13. بما أن 13 أقل من 32 سنقفز إلى العقدة اليسارية للجذر. وهناك نجد أن 13 أقل من 16، لذلك سنستمر في الاتجاه نحو العقدة اليسارية لـ 16. وهناك نجد أن 13 أكبر من 1، وعليه ستصبح الـ 13 هي العقدة اليمينية للـ 1.
7. وبالمثل قم بعمليات الإضافة كما سبق حتى تصل إلى النهاية، أي قبل الرقم 53.
8. الرقم 53. بما أن 53 أكبر من 32 سوف نقفز إلى العقدة اليمينية للجذر. وهناك نجد أن 53 أكبر من 34، لذلك نستمر في الاتجاه نحو العقدة اليمنية لـ 34. وهناك نجد أن 53 أقل من87 لذلك سنتحرك بالاتجاه نحو العقدة اليسارية لـ 87. وهناك سنجد أن 53 أكبر من 41 لذلك سنقفز إلى العقدة اليمينية لـ 41. وبما أن العقدة اليمينية لـ 41 فارغة، فستصبح 53 هي العقدة اليمينية لـ 41.

ما قمنا به إلى الآن ينبغي أن يعطيك فكرة عن كيفية عمل الشجرة الثنائية. يجب أن تعرف أن:

1. ربط العقد بعقد في الشجرة الثنائية هو عملية ذات طبيعة واحد إلى واحد، أي أنه: لا يمكن أن يتم الإشارة إلى أي عقدة بأكثر من عقدة واحدة.
2. ويمكن أن تشير أي عقدة إلى عقدتين فرعيتين مختلفتين على الأكثر.

نلاحظ هنا في الشجرة الثنائية أعلاه أن هناك بعض العقد التي يكون فيها المؤشر اليساري واليميني فارغين، أي أنه: ليس لديهم عقد فرعية مرتبطة بهم. فالعقد 5،14،18،19،23،24،41 كلها لا تملك عقداً مرتبطة يسارية.

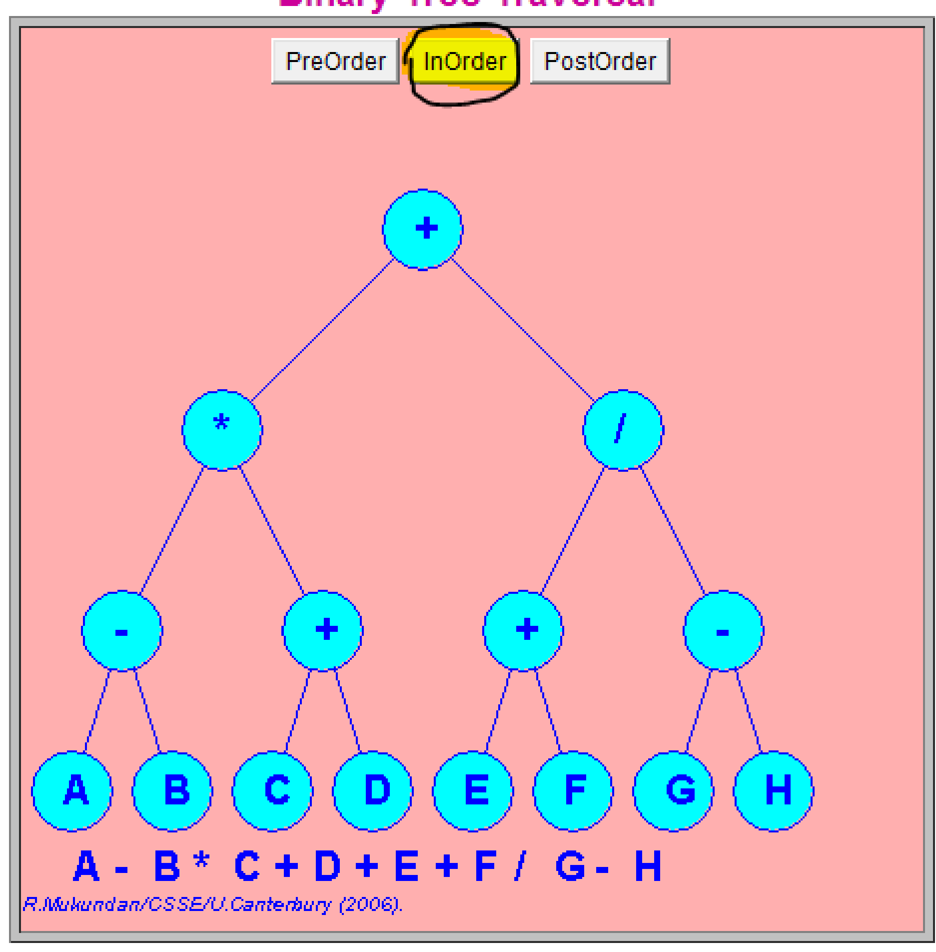
هناك ثلاث طرق شائعة لعرض بيانات الشجرة الثنائية. عملية عرض محتويات الأشجار تعرف أيضاً بالتنقل transversal. هناك ثلاث طرق للتنقل، هي التنقل الترتيبي inorder، والتنقل السابق preorder والتنقل اللاحق postorder. وسنقوم فيما يلي بتوضيح كل طريقة:

التنقل السابق PREORDER:

1. قم بزيارة الجذر.
2. انتقل إلى الورقة اليسارية بأسلوب التنقل السابق.
3. انتقل إلى الورقة اليمينية بأسلوب التنقل السابق.

التنقل الترتيبي INORDER:

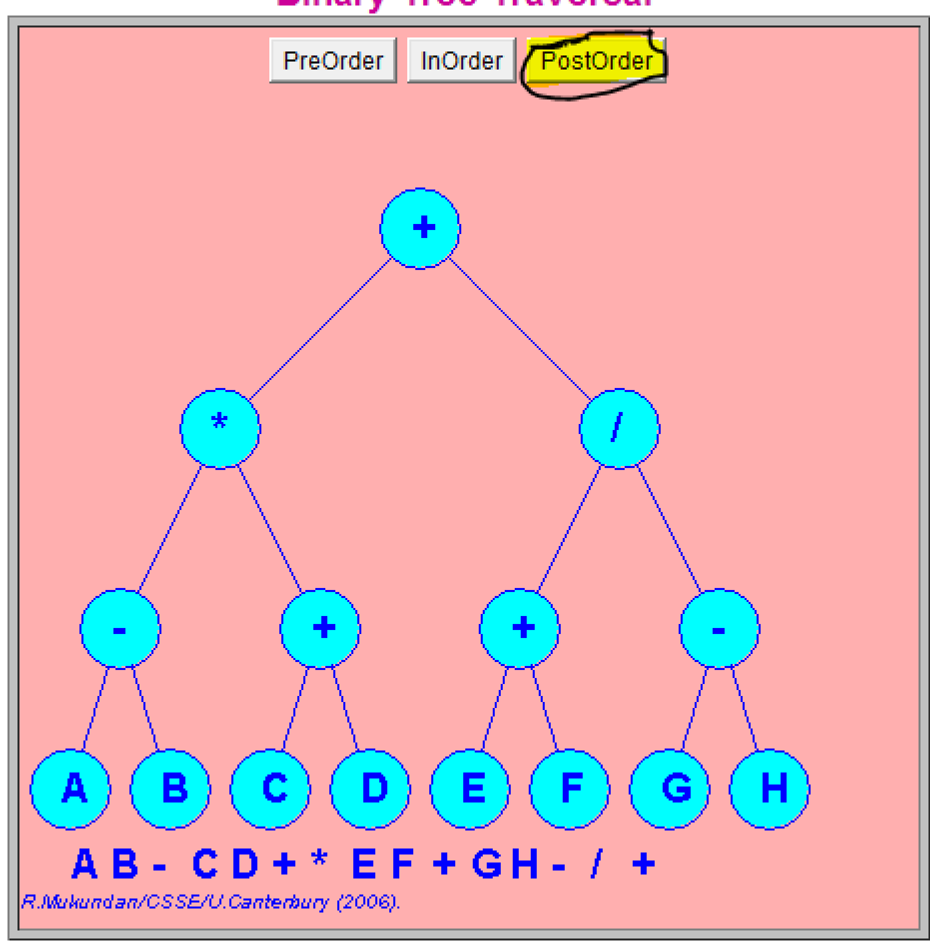
1. انتقل إلى الورقة اليسارية بأسلوب التنقل الترتيبي.
2. قم بزيارة الجذر.
3. انتقل إلى الورقة اليمينية بأسلوب التنقل الترتيبي.



التنقل اللاحق POSTORDER:

1. انتقل إلى الورقة اليسارية بأسلوب التنقل اللاحق.
1. انتقل إلى الورقة اليمينية بأسلوب التنقل اللاحق.
2. قم بزيارة الجذر.





إن كتابة الشفرة لهذه الطرق الثلاث هو أمر بسيط إذا فهمنا الطبيعة التعاودية recursive للشجرة الثنائية. ونقصد بالطبيعة التعاودية أن كل عقدة في الشجرة يمكن ننظر إليها على أنها شجرة ثنائية قائمة بنفسها. إن الفرق الوحيد في التنقل يكمن في ترتيب عرض البيانات.

أما عملية الحذف من الشجرة الثنائية فهي أكثر صعوبة قليلاً من حيث الفهم. ولغاية الآن فقط تذكر أنه لحذف أي عقدة، سيتم استبدالها مع العقدة التالية لها بالترتيب inorder. سأوضح كل شيء بعد شفرة الشجرة الثنائية.

الآن بعد أن تعرفنا على أساسيات التعامل مع الشجرة الثنائية، دعونا ننتقل إلى الشفرة المصدرية.

```
#include <iostream>

using namespace std;

#define YES 1
#define NO 0

class Tree
{
	private:
		struct leaf
		{
			int data;
			leaf *l;
			leaf *r;
		};
		leaf *p;

	public:
		Tree();
		~Tree();
		void destruct(leaf *q);
		Tree(Tree& a);
		void findparent(int n,int &found,leaf* &parent);
		void findfordel(int n,int &found,leaf *&parent,leaf* &x);
		void add(int n);
		void transverse();
		void in(leaf *q);
		void pre(leaf *q);
		void post(leaf *q);
		void del(int n);
};

Tree::Tree()
{
	p = NULL;
}

Tree::~Tree()
{
	destruct(p);
}
```

```
void Tree::destruct(leaf *q)
{
     if(q!=NULL)
     {
          destruct(q -> l);
          del(q -> data);
          destruct(q -> r);
     }
}
void Tree::findparent(int n,int &found,leaf *&parent)
{
     leaf *q;
     found = NO;
     parent = NULL;

     if(p == NULL)
          return;

     q = p;
     while(q != NULL)
     {
          if(q -> data == n)
          {
               found = YES;
               return;
          }
          if(q -> data > n)
          {
               parent = q;
               q = q -> l;
          }
          else
          {
               parent = q;
               q = q -> r;
          }
     }
}
```

```
void Tree::add(int n)
{
	int found;
	leaf *t,*parent;
	findparent(n, found, parent);
	if(found == YES)
		cout << "\nSuch a Node Exists";
	else
	{
		t = new leaf;
		t -> data = n;
		t -> l = NULL;
		t -> r = NULL;

		if(parent == NULL)
			p = t;
		else
			parent -> data > n ? parent -> l = t : parent -> r = t;
	}
}

void Tree::transverse()
{
	int c;
	cout << "\n1.InOrder\n2.Preorder\n3.Postorder\nChoice: ";
	cin >> c;
	switch(c)
	{
		case 1:
			in(p);
			break;
		case 2:
			pre(p);
			break;
		case 3:
			post(p);
			break;
	}
}

void Tree::in(leaf *q)
{
	if(q != NULL)
	{
		in(q -> l);
		cout << "\t" << q -> data << endl;
		in(q -> r);
	}

}
```

```
void Tree::pre(leaf *q)
{
	if(q != NULL)
	{
		cout << "\t" << q -> data << endl;
		pre(q -> l);
		pre(q -> r);
	}
}

void Tree::post(leaf *q)
{
	if(q != NULL)
	{
		post(q -> l);
		post(q -> r);
		cout << "\t" << q -> data <<endl;
	}

}

void Tree::findfordel(int n,int &found,leaf *&parent,leaf *&x)
{
	leaf *q;
	found = 0;
	parent = NULL;
	if(p == NULL)
		return;

	q = p;
	while(q != NULL)
	{
		if(q -> data == n)
		{
			found = 1;
			x = q;
			return;
		}
		if(q -> data > n)
		{
			parent = q;
			q = q -> l;
		}
		else
		{
			parent = q;
			q = q -> r;
		}
	}
}
```

```
void Tree::del(int num)
{
	leaf *parent,*x,*xsucc;
	int found;

	// If EMPTY TREE
	if(p == NULL)
	{
		cout << "\nTree is Empty";
		return;
	}
	parent = x = NULL;
	findfordel(num, found, parent, x);
	if(found == 0)
	{
		cout << "\nNode to be deleted NOT FOUND";
		return;
	}

	// If the node to be deleted has 2 leaves
	if(x -> l != NULL && x -> r != NULL)
	{
		parent = x;
		xsucc = x -> r;

		while(xsucc -> l != NULL)
		{
			parent = xsucc;
			xsucc = xsucc -> l;
		}
		x -> data = xsucc -> data;
		x = xsucc;
	}

	// if the node to be deleted has no child
	if(x -> l == NULL && x -> r == NULL)
	{
		if(parent -> r == x)
			parent -> r = NULL;
		else
			parent -> l = NULL;

		delete x;
		return;
	}
```

```
	// if node has only right leaf
	if(x -> l == NULL && x -> r != NULL )
	{
		if(parent -> l == x)
			parent -> l = x -> r;
		else
			parent -> r = x -> r;

		delete x;
		return;
	}

	// if node to be deleted has only left child
	if(x -> l != NULL && x -> r == NULL)
	{
		if(parent -> l == x)
			parent -> l = x -> l;
		else
			parent -> r = x -> l;

		delete x;
		return;
	}
}

int main()
{
	Tree t;
	int data[] = {32,16,34,1,87,13,7,18,14,19,23,24,41,5,53};
	for(int iter = 0 ; iter < 15 ; i++)
		t.add(data[iter]);

	t.transverse();
	t.del(16);
	t.transverse();
	t.del(41);
	t.transverse();
	return 0;
}
```

المخرجات OUTPUT:

```
1.InOrder
2.Preorder
3.Postorder
Choice: 1
1
5
7
13
14
```

```
16
18
19
23
24
32
34
41
53
87

1.InOrder
2.Preorder
3.Postorder
Choice: 2
32
18
1
13
7
5
14
19
23
24
34
87
41
53

1.InOrder
2.Preorder
3.Postorder
Choice: 3
5
7
14
13
1
24
23
19
18
53
87
34
32
Press any key to continue.
```

ملاحظة NOTE:
قد تظهر أخطاء أثناء التشغيل لهذه الشفرة إذا استخدمت Visual C++. لذلك قم بترجمته باستخدام Turbo C++.

أنه وبمجرد نظرنا إلى المخرجات يمكننا أن ندرك أنه يمكن أن نطبع كامل الشجرة بترتيب تصاعدي باستخدام التنقل الترتيبي inorder. وفي حقيقة الأمر فإن الأشجار الثنائية تستخدم في عمليات البحث [أشجار البحث الثنائية Binary Search Trees {BST}] وكذلك في عملية الترتيب Sorting. إن إضافة جزء البيانات يبدو واضحاً. فقط عملية الحذف تحتاج إلى قليل من التوضيح.

لحذف البيانات هناك عدد من الحالات التي يجب أخذها بعين الاعتبار:

1. إذا كانت الورقة leaf المراد حذفها غير موجودة.
2. إذا لم يكن للورقة المراد حذفها أوراق فرعية.
3. إذا كان للورقة المراد حذفها ورقة فرعية واحدة.
4. إذا كان للورقة المراد حذفها ورقتين فرعيتين.

الحالة الأولى CASE 1: التعامل مع هذه الحالة أمر بسيط، كل ما يتوجب علينا هو ببساطة عرض رسالة خطأ.
الحالة الثانية CASE 2: بما أن العقدة المراد حذفها لا تملك عقداً فرعية، فإنه يجب تحرير الذاكرة المخصصة لهذه العقدة، ويجب أن نجعل قيمة المؤشر اليساري أو المؤشر اليميني للعقدة الأم فارغاً NULL. ومسألة تحديد أي من هذه المؤشرات سيكون NULL سيتوقف على كون العقدة المراد حذفها يسارية أم يمينية بالنسبة للعقدة الأم.

الحالة الثالثة CASE 3: وفي الحالة الثالثة يتوجب علينا فقط أن نعدل مؤشر الأب للورقة المراد حذفها بحيث يشير بعد عملية الحذف إلى العقدة الابن للعقدة المراد حذفها.

الحالة الرابعة CASE 4: الحالة الأخيرة التي يكون فيها للورقة المراد حذفها ورقتين فرعيتين هي أكثر الحالات تعقيداً. المنطق الكلي لهذه الحالة يكمن في تحديد التالي الترتيبي inorder، ومن ثم نسخ بياناته وتحويل المشكلة إلى عملية حذف بسيطة لعقدة تملك ورقة واحدة أو لا شيء من الأوراق.
إذا نظرنا في البرنامج المذكور أعلاه ... (وبالإشارة إلى الشجرة السابقة أيضاً) فإنه عند حذفنا لـ 16 فإننا نبحث عن التالي الترتيبي. لذلك نقوم ببساطة بتحويل القيمة البيانية 5 وحذف العقدة التي تحمل القيمة 5 كما هو موضح في الحالات 2 و3.

هذا كل شيء.

وتستخدم الأشجار الثنائية لأشياء أخرى مختلفة بما في ذلك خوارزميات الضغط Compression Algorithms، والبحث الثنائي Binary Searching، والترتيب Sorting وما إلى ذلك. وهناك الكثير من خوارزميات هوفمان، شانون، فانو وخوارزميات ضغط أخرى تستخدم الأشجار الثنائية.

عاشراً: الاتصال بي Contact me

يرجى التواصل على الإيميلات التالية:

fatta7mail@yahoo.com
fatta7mail@gmail.com
fatta7mail@facebook.com